The Pragmatic Programmers

Seven Languages in Seven Weeks:
A Pragmatic Guide to Learning Programming Languages

7つの言語 7つの世界

Ruby, Io, Prolog, Scala, Erlang, Clojure, and Haskell

Bruce A. Tate 著
まつもとゆきひろ 監訳／田和勝 訳

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

Original English language title:
Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages
by Bruce A. Tate
Published by The Pragmatic Programmers, LLC

Copyright © 2010 The Pragmatic Programmers, LLC
Translation Copyright © 2011 Ohmsha, Ltd.

All rights reserved.
No part of this book may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

本書を発行するにあたって，内容に誤りのないようできる限りの注意を払いましたが，本書の内容を適用した結果生じたこと，また，適用できなかった結果について，著者，出版社とも一切の責任を負いませんのでご了承ください．

本書に掲載されている会社名・製品名は一般に各社の登録商標または商標です．

本書は，「著作権法」によって，著作権等の権利が保護されている著作物です．本書の複製権・翻訳権・上映権・譲渡権・公衆送信権（送信可能化権を含む）は著作権者が保有しています．本書の全部または一部につき，無断で転載，複写複製，電子的装置への入力等をされると，著作権等の侵害となる場合がありますので，ご注意ください．

本書の無断複写は，著作権法上の制限事項を除き，禁じられています．本書の複写複製を希望される場合は，そのつど事前に下記へ連絡して許諾を得てください．

- オーム社開発部「7つの言語7つの世界」係宛，E-mail（kaihatu@ohmsha.co.jp）または書状，FAX（03-3293-2825）にて

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

複数のパラダイムを理解していると設計能力が大幅に強化されるので，私はそうした知識の得られる良書をいつも探している．本書は有名な複数のパラダイムをコンパクトに解説している．著者はいくつもの言語を学習し使ってきた経験がある．本書はその著者の経験を追体験できる．お勧めの一冊だ．

➤ *Dr. Venkat Subramaniam*
Agile Developer, Inc. 受賞歴のある著者・創業者

プログラマにとって，新しいプログラミング言語，パラダイム，技法を直接体験することの大事さはいくら強調しても強調しすぎることはない．本書は，7つの重要かつ多様な言語を，コンパクトに，しかし決して表面的にならないように紹介しつつ，各言語の強みと存在理由を示すという素晴らしい仕事をやってのけている．「新しい概念を学びたい」あるいは「本格的に学ぶ言語を決める前に新しく登場した言語をいくつか評価したい」プログラマにとって，本書はさまざまな料理を小皿に分けて食べさせてくれるビュッフェのような役割を果たしてくれるだろう．

➤ *Antonio Cangiano*
IBM ソフトウェアエンジニア／技術エバンジェリスト

本書は，プログラミング言語の短期集中コースだ．かなりのペースで説明しているので落伍しないように覚悟して読む必要がある．著者は率直な語り口で説明しており，魅力的で読みがいのある内容になっている．熱心なプログラマなら大いに楽しめるだろう．新しい言語を学びたい，自分の脳の限界に挑戦してみたい，プログラマとしてのスキルアップを図りたいと考えているなら，本書はうってつけだ．読んでがっかりすることはないだろう．

➤ *Frederic Daoud*
Stripes ...and Java Web Development Is Fun Again，*Getting Started with Apache Click* 著者

「7つの言語を手早く学んでから本格的に学ぶ言語を決めたい」あるいは「一般に，プログラミングに関して，自分の脳の限界に挑戦してみたい」とお考えなら，本書に勝る本はない．個人的には，コンピュータサイエンスを専攻していた大学時代に，プログラミング言語の概説講義をのんびり聞き流していたときのことを思い出した．ただし，この本は決してのんびり斜め読みできるような内容ではない．積極的に取り組む必要がある．そうすれば，脳への刺激になるうえに極めて実用的な本だと気づくだろう．

➤ *Matt Stine*
科学研究アプリケーション開発部門（St. Jude Children's Research Hospital） グループリーダー

コンピュータサイエンス専攻の学生時代，ソフトウェア開発者にはなりたくないとずっ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

と思っていたが，結局そうなってしまった．この本は私のプログラムに対する考え方を広げてくれ，自分がプログラミングを好きな理由を思い出させてくれた．

➤ *Travis Kaspar*
ノースロップ・グラマン ソフトウェア技術者

25 年間，さまざまなハードウェアとソフトウェア言語を経験してきたが，この本を読んで，言語の強みと弱みを客観的に評価する方法が分かってきた．何より，7 つのなかから 1 つの言語を選択して実際の業務に使ってもよいと感じている．

➤ *Chris Kappler*
Raytheon BBN Technologies 上級サイエンティスト

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

母に捧ぐ

2009 年 12 月から 2010 年 4 月までの 5 か月間は，筆者の人生の中でも最も苦しい時期だった．まだ 47 歳の兄が心臓の緊急バイパス手術を受けた．誰も兄がそのような状態になっていることにまったく気づかなかった（手術は成功し，以降何事もなく元気でやっている）．3 月後半には，姉が乳ガンと診断された．最大のショックは，3 月初めに母が末期ガンを宣告されたことだった．それから数週間後，母は亡くなった．

お察しのとおり私は，効率的なまでに情け容赦ないガンという病気による，衝撃的で予想もしなかった母の死に直面し，深い悲しみのなかにいる．誰しもそうなるだろう．しかし奇妙なことに，この経験は悲しみ一色ではなかった．母は素晴らしい人生を生きた．家族とのつながりも強く充実したものだったし，信念をもって自分の望む場所で生きた人だった．

母 Lynda Lyle Tate は，自分の創造的エネルギーを水彩画に注いだ．彼女は主に，自分の主催する Madison Avenue アートギャラリーと自分のクラスで教えていた．私は家を出て独立する前に，母に何度か絵の手ほどきを受けたことがある．技術職に携わる者にとって，絵を描くことはいつもとまどうような経験だった．真っ白なキャンバスに自らの傑作をイメージするわけだが，実際に絵が形になるにつれて，いつのまにか元のイメージからどんどん離れていってしまう．これはもう直せっこないと落ち込んでいると，肩越しに見ていた母がアドバイスをくれた．彼女があざやかな筆さばきで，奥行きをつけるために影を入れ，明るい色ではっきりとした線や細かい線を加えると，まったく失敗作などではなかったのだと気づかされたものだ．才能ある人が少し筆を入れるだけで，大失敗も同然だった自分の作品が息を吹き返した．私は興奮して勝ち誇ったように両手を上げ，クラス全員に自分の作品を得意げに吹聴した．実はクラスの他の生徒たちも皆同じように喜びで胸がいっぱいだったのだが，そのことには気づきもせずに．

しばらくして，私は，母が別の種類のキャンバスにも取り組んでいたことを知った．教会を通して，絵の教師という自分の職業を通して，母は絶望した人たちを見つけては手をさしのべていた．伴侶を亡くした人もいれば，うまくいっていない夫婦もいた．母はそうした人たちをクラスに連れてきては，絵の具と紙を使って，彼らの閉ざされた心のドアを少し開いてあげた．私は最後の 1 週間を母とともに過ごした．その間，自分の先生が亡くなってしまうことに打ちひしがれた生徒たちが次から次へと病室にやってきたが，母は完璧なジョークを飛ばしたり，やさしい言葉をかけたりして，慰めにきた人たちを逆に慰めていた．師によって正しい方向に導かれ，素晴らしい仕事をした人たち．私は母の病室で，母の助けを借りて人生というキャンバスに絵を描いた人たちに会う機会を得た．それは謙虚な気持ちにさせられる経験だった．

母にこの本を捧げると言うと，母は「それはうれしいけれど，私はコンピュータとは縁がないからねえ」と言った．確かにそのとおりだ．Windows と聞いただけで彼女はお手上げだろう．でもね，母さん，あなたは僕とは誰よりも深い縁で結ばれていた．あなたのここぞというときの励ましの言葉に勇気をもらい，あなたの創造性を愛する心が今の僕を

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

形づくり，あなたの命に対する情熱と愛は今でも僕を導いている．あの病室での経験のことを考えると，僕は少し安らぎ，少し強い気持ちにならざるを得ない．なぜなら，僕自身も，あなたという師の助けを借りて人生というキャンバスに絵を描いた一人なのだから．

この本を愛を込めて母 Lynda Lyle Tate（1936〜2010 年）に捧げる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

謝辞

本書は私のこれまでの著作のなかで最も苦労させられた本である．と同時に最も満足感の大きい本でもある．これも，さまざまな形で手伝ってくれた方々のおかげだ．まず家族に感謝したい．Kayla，Julia，君たちの文章力には驚いた．君たちの能力は，自身の想像をはるかに超えている．Maggie，君は私の喜びであり，インスピレーションの源だ．

Ruby コミュニティでは，まず Dave Thomas に感謝したい．彼のおかげで Ruby という言語に出会うことができた．Ruby は私のキャリアを根底からくつがえし，プログラミングの楽しみを再発見させてくれた．また Matz の友情に感謝したい．彼は本書に掲載したインタビューでいろいろな質問に答えてくれた．また，Ruby の生まれ故郷である日本の松江にご招待いただいた．あの経験は，Matz が思っているよりずっと大きなインスピレーションを私に与えてくれた．Charles Nutter，Evan Phoenix，Tim Bray の 3 人にも感謝したい．本書では Ruby に関するさまざまなトピックを取り上げた．これらは退屈にも思えたが，彼らが洗練された明解な文章にしてくれた．ありがとう．

Io コミュニティでは，Jeremy Tregunna に感謝したい．彼のおかげで Io という言語に対する理解が深まった．また彼は本書に掲載するサンプルプログラムも提供してくれた．レビューも最高のものだった．タイミング良くレビューを提供してくれたので，しっかりとした内容にすることができた．Io の作者である Steve Dekorte にも感謝したい．市場での評価がどうであれ，彼は素晴らしい言語を作り上げた．並列性機能は感動的だし，言語に美しさが組み込まれている．私は Io という言語の正しさを心底認めることができる．私のような初心者のインストール環境の修正を手伝ってくれた．配慮の行き届いたレビューをしてもらい，インタビューにも丁寧に応じてくれた．おかげで Io という言語のエッセンスをつかむことができた．Io は本書のベータ読者たちの心をとらえ，多くの読者のお気に入りの言語になった．

Prolog コミュニティでは，インタビューで貴重な体験を語ってくれた Brian Tarbox に感謝したい．イルカの研究プロジェクト（科学ドキュメンタリ番組の NOVA でも特集された）の体験談は，間違いなく読者が Prolog を直感的に理解するための効果的なエピソードとなった．Joe Armstrong には特に感謝したい．彼が提供してくれたフィードバックによって，Prolog の章だけでなく本書全体が形をなすようになった．また，地図の塗り分けの例や append のアイデアなど，いろいろと助けてもらった．これらの例は，最適な例として最適な場所で使わせてもらった．

Scala コミュニティでは，私の友人である Venkat Subramaniam に感謝したい．彼の Scala に関する書籍は，情報量が豊富でかつ分かりやすい．大いに活用させてもらった．彼にはレビューにも参加してもらい，執筆途中でちょっとした質問に答えてもらった．彼が時間を割いて質問に答えてくれたおかげで無駄に苦しまずに済み，執筆作業に専念できた．Martin Odersky にも感謝したい．彼とは面識がなかったにもかかわらず，いろいろな考え方を本書のために提供してくれた．Scala はユニークで思い切ったアプローチによって，関数型プログラミングとオブジェクト指向パラダイムを統合した言語だ．その

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

努力は高く評価されている．

Erlang コミュニティでは，やはり Joe Armstrong に感謝したい．彼の好意とエネルギーのおかげで，本書のアイデアを形にすることができた．彼は，分散型の耐障害性システムのしかるべき構築方法を根気強く奨励しており，それが功を奏している．私には，Erlang の「クラッシュさせろ」の哲学が，本書で紹介した他の言語が採用しているどのアイデアよりも理にかなっているように思える．このアイデアがもっと広く採用されることを願っている．

Clojure コミュニティでは，Stuart Halloway に感謝したい．彼はレビューにも参加してもらった．また，読者により良い本を届けるために頑張るよう激励してくれた．彼の Clojure に対する深く正確な理解と直感のおかげで，何が重要なのかを理解することができた．彼の著作は，Clojure の章で大いに参考にさせてもらったし，他の章でも問題に対する別のアプローチの方法を思いつかせてくれた．彼のコンサルティング業におけるアプローチは高く評価されている．彼の手法は，強く求められているシンプルさと生産性を業界にもたらした．Rich Hickey にも感謝したい．彼は，Clojure を開発するときの思慮深いアイデアと，Lisp 方言であることの意味を教えてくれた．Clojure で採用されている概念は極めて急進的だが，同時に実用的でもある．Lisp を再び革命的な言語にする方法を見つけた彼に「おめでとう」と言いたい．

Haskell コミュニティでは，Haskell が開発された経緯について話を聞かせてくれた Phillip Wadler に感謝したい．彼は私と同様教えることに情熱を持っており，教えるのが大変うまい．Simon Peyton-Jones にも感謝したい．彼とのインタビューは楽しかった．彼はさまざまな洞察を与えてくれたし，読者にも独自の観点を提供してくれた．

本書のレビューアである Vladimir G. Ivanovic, Craig Riecke, Paul Butcher, Fred Daoud, Aaron Bedra，David Eisinger，Antonio Cangiano，Brian Tarbox に感謝したい．彼らは素晴らしい仕事をしてくれた．私がこれまでに仕事したなかで最も効果的なレビューチームだった．おかげで本書の内容に強固な基盤が形成された．書籍をこれほど詳細にレビューするのは，割に合わない大変な作業であることを私は理解している．技術書を愛する人たちは，レビューアの皆さんの作業に感謝している．出版業界は皆さんなくしては存在しえない．

本書での言語の選択およびプログラミング哲学に関して，さまざまな考え方を提示してくれた皆さんにも感謝したい．Neal Ford，John Heintz，Mike Perham，Ian Warshak は，さまざまな場面で，重要な貢献をしてくれた．彼らとの会話のおかげで，読者から見た私は現実よりも賢く見えたのではないかと思っている．

ベータ読者の皆さん，本書を査読してくれてありがとう．皆さんのおかげで私は執筆作業を諦めずに継続できた．皆さんからいただいたコメントを読むと，単に斜め読みしているのではなく，各言語にきちんと取り組んでいることがよく分かった．これまで何百というコメントに基づいて内容を修正してきたし，これからも本書が読まれ続ける限り，皆さんのコメントを反映させていきたいと思っている．

最後に，Pragmatic Bookshelf のスタッフに心から感謝の意を表する．Dave Thomas と Andy Hunt の二人は，プログラマとして，また著者としての私のキャリアに，とてつもなく大きな影響を与えた．Pragmatic Bookshelf という出版社が存在したおかげで，また

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

こうして本を書くことができた．本書の類書で，思ったほど読者に受けず黒字化していない書籍はいくらでもある．出版チームのすべてのメンバーに感謝したい．Jackie Carterは，やさしく筆者を誘導してくれた．彼女のこうした姿勢が本書には必要だった．彼女との会話は楽しかった．彼女も同じように楽しんでくれたことを願っている．裏方役に徹して本書の体裁を整え，筆者の悪い癖を直そうとしてくれたチームの皆さんに感謝したい．編集者のKim Wimpsett，索引作成担当のSeth Maislin，組版担当のSteve Peter，プロデューサーのJanet Furlow，皆さんがいなければ本書は現在のような立派な形にはなっていないだろう．ありがとう．

いつものことだが，この素晴らしいチームのメンバーで名前を挙げなかった皆さん，本当に申し訳ない．すべては筆者の責任である．決して故意ではないのでお許し願いたい．

最後に，読者の皆さんに感謝したい．筆者は印刷された本は価値があると考えているし，皆さんが同じように考えているからこそ，情熱の赴くままに書き続けることができる．

Bruce Tate

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:09:39 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:09:39 under the license of University of Tokyo

執筆中の著作『プルーストを読めば良いプログラマになれる理由』より
Joe Armstrong（Erlang の作者）
「Gmail のエディタが引用符の起こしと受けを正しく入力させてくれないのですが」
「けしからん．教養のないプログラマと退廃的な文化の兆候だ」
「どうすればよいでしょうか」
「プルーストの『失われたときを求めて』を読んでいることをプログラマの採用条件に含める必要があるだろう」
「7 巻すべてですか」
「そうだ．7 巻すべてだ」
「それで句読点の使い方が改善され，引用符の使い方が正しくなるでしょうか」
「それは分からないが，良いプログラマにはなるだろう．禅的な考え方だよ……」

プログラミングを学ぶことは，泳ぎを覚えるのに似ている．いくら理論を学んでも，プールに飛び込んで，息をしようと喘ぎながら水の中で手足をばたつかせる経験には代えられない．最初水中に沈んだときはパニック状態になるが，水面にひょいと顔を出して空気を飲み込むと，うれしい気分になる．「自分は泳げる」と思えるからだ．少なくとも，私は泳ぎを覚えるときそんな風に感じた．

プログラミングにも同じことが言える．学び始めが一番難しい．水に飛び込む気持ちになるには，良い先生が必要だ．

Bruce Tate はそんな先生だ．本書はプログラミングを学ぶときに最も困難な時期，すなわち最初のとっかかりの機会を提供するものだ．

興味のある言語のインタープリタまたはコンパイラをダウンロードおよびインストールする面倒な作業を何とか終えたとしよう．次に何をすべきだろうか．最初のプログラムはどのような内容だろうか．

Bruce はこの質問に見事に答えてくれる．「本書に掲載したプログラムまたはプログラムの一部を単に入力して，本書と同じ結果が得られるかどうか確認してみよ．自分でプログラムを書こうなどと思ってはいけない．本書の例を単に写すだけでよい．自信がついてくると，自分のプログラミングプロジェクトに挑戦できるようになる」．

何か新しいスキルを取得するときに実行すべき最初のステップは，自分で新しく何かを行えるようになることではなく，他の人が既にやってみせたことを自分でも再現できるようになることである．これが，新しいスキルを身につけるための最短経路だ．

新しい言語でプログラミングを始めるときに行うのは，その言語を構成している基本原理を理解するための難しい練習をこなすことなどではない．むしろ，セミコロンやカンマを正しい位置に指定するとか，何かを間違えたときにシステムが出力する奇妙なエラーメッセージを理解するといった単純なことだ．プログラムを入力し，コンパイラエラーをなくすという面倒な作業を終えて初めて，さまざまな構文の意味を考え始めることができるのである．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

プログラムの入力と実行という機械的な作業を終わらせてしまえば，あとはゆったりくつろげる．残りの作業であるコードの解釈は潜在意識がやってくれるからだ．脳が意識してセミコロンの位置を決めている間に，潜在意識が表面的な構文の下に隠れている深い意味を理解する．そしてある日突然，論理プログラムの深い意味や特定の言語が特定の構文を備えている理由が分かるのだ．

多くの言語について少しだけ知っていると有用なスキルになる．筆者は，特定の問題を解くために，Python や Ruby を少しだけ理解する必要があると気づくことがよくある．また，インターネットからダウンロードするプログラムはさまざまな言語で書かれており，個人で使うには少し手を入れなければならないことが多い．

各言語にはそれぞれ，イディオム，長所，短所がある．さまざまなプログラミング言語を学ぶことによって，自分が興味を持っている種類の問題を解くのに最適な言語が分かるようになる．

喜ばしいことに，本書の著者である Bruce のプログラミング言語の好みは多岐にわたっている．彼は Ruby のような確立された言語だけでなく，Io のようなあまり評価されていない言語も取り上げている．プログラミングとは結局，理解することであり，理解できるかどうかはどれだけアイデアの引き出しがあるかにかかっている．したがって，新しい言語を直接体験することは，プログラミングが何たるかをより深く理解するために欠かせない．

禅の指導者は，数学が出来るようになりたければラテン語を勉強せよと言うだろう．プログラミングでも同じだ．オブジェクト指向プログラミングの本質を深く理解するには，論理プログラミングや関数型プログラミング（FP）を勉強する必要がある．関数型プログラミングに上達したければ，アセンブラを勉強する必要がある．

筆者がプログラマとして経験を積んでいた頃，プログラミング言語を比較した本は人気があったが，その大半は学術的な本で，言語の実際の使い方に関する実践的な情報は得られなかった．そうした状況が，その当時のテクノロジを反映していた．当時は，言語の概念について読むことはできたが，それを実際に試してみるのは事実上不可能だった．

現在では，概念について読むことができるだけでなく，実際に試してみることができる．泳ぐのは楽しいだろうかと思いながらプールサイドで突っ立っているのと，プールに飛び込んで水を楽しむのとでは雲泥の差がある．読むのと実際に試すのも，同じくらい大きな違いがある．

本書を心からお勧めする．皆さんが私と同じように本書を楽しまれるよう願っている．

2010 年 3 月 2 日

Erlang の作者
Joe Armstrong
ストックホルムにて

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

目次

母に捧ぐ v

謝辞 vii

序文 xi

第 1 章 はじめに 1
1.1 本書のねらい 1
1.2 この本で扱う言語 2
1.3 この本を買ってください 4
1.4 この本を買わないでください 5
1.5 終わりに 7

第 2 章 Ruby 9
2.1 簡単な歴史 9
2.2 1 日目：乳母を探す 11
2.3 2 日目：空からふわふわ舞い降りる 18
2.4 3 日目：真剣な変更 30
2.5 Ruby のまとめ 37

第 3 章 Io 41
3.1 Io とは 41
3.2 1 日目：学校をさぼって，気ままに過ごす 42
3.3 2 日目：ソーセージキング 54
3.4 3 日目：パレードとその他の奇妙な場所 62
3.5 Io のまとめ 70

第 4 章 Prolog 73
4.1 Prolog について 74
4.2 1 日目：素晴らしいドライバー 74
4.3 2 日目：Wapner 判事まで 15 分 85
4.4 3 日目：ラスベガスをぶっとばせ 95
4.5 Prolog のまとめ 105

第 5 章 Scala 109
5.1 Scala について 109
5.2 1 日目：丘の上の城 112

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

5.3 2 日目：藪を刈り込む 126
5.4 3 日目：快刀乱麻を断つ 138
5.5 Scala のまとめ 146

第 6 章 Erlang 151
6.1 Erlang について 151
6.2 1 日目：外見は普通 155
6.3 2 日目：変身 164
6.4 3 日目：赤い錠剤（The Red Pill） 174
6.5 Erlang のまとめ 184

第 7 章 Clojure 189
7.1 Clojure とは 189
7.2 1 日目：ルークの修行 191
7.3 2 日目：マスター・ヨーダとフォース 207
7.4 3 日目：悪を見逃すな 219
7.5 Clojure のまとめ 226

第 8 章 Haskell 231
8.1 Haskell とは 231
8.2 1 日目：論理的 232
8.3 2 日目：スポックの大きな強み 246
8.4 3 日目：精神融合 254
8.5 Haskell のまとめ 267

第 9 章 全体のまとめ 271
9.1 プログラミングモデル 271
9.2 並行性 274
9.3 プログラムを構成する要素 276
9.4 自分なりの表現を見いだす 278

付録 A 参考資料 279

監訳者あとがき 281

索引 283

著者・監訳者・訳者について 287

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第 1 章

はじめに

人が言葉を学ぶ理由はさまざまだ．最初の言語，つまり母語は生きるために学ぶ．それは日々生活していくための道具となる．外国語を学ぶ理由もさまざまだ．キャリアアップのためかもしれないし，変化する環境に適応するためかもしれない．場合によっては，学ばなければならないからではなく，学びたいから，新しい言語をマスターしようと決心することもあるだろう．外国語を学ぶと新しい世界が開ける．新しい言語を学ぶごとに物の考え方が形成されることを認識したうえで，何かの啓示を得るために言語を学ぶ人さえいるだろう．

プログラミング言語にも同じことが言える．本書では 7 つの異なる言語を紹介する．といっても，母親が毎朝苦い薬を飲ませるようなやり方はしない．読者の皆さんに深い洞察を与え，プログラミングに対する見方が変わるような経験をしてもらうことが目的だ．本書を読んでプログラミング言語の専門家になることはできないが，お決まりの “Hello, World” プログラムよりは高度なことをお教えするつもりだ．

1.1 本書のねらい

多くの場合，筆者が新しいプログラミング言語またはフレームワークを学ぶときは，短時間で読める対話型のチュートリアルを探す．目的は，管理された環境で言語を経験することだ．必要ならスクリプトを実行して探索することもできるが，基本的には，脳に軽い刺激をもらい，シンタックスシュガーを垣間見て，主要な概念を把握することができればよい．

しかし，こうしたチュートリアルで満足できることはまずない．既に知っている知識に少し付け足した程度ではなく，あるプログラミング言語の雰囲気を本当に味わいたければ，短いチュートリアルで足りることは決してない．深くしっかりと学ぶ必要がある．本書では，そうした経験を 1 回ではなく 7 回してもらうつもりだ．本書を読めば，次の質問に答えられるようになる．

- 型付けモデルは？ 型付けには，強い型付け（Java）と弱い型付け（C），静的型付け（Java）と動的型付け（Ruby）がある[*1]．本書に登場する言語は強い型付けのほうに近いが，静的型付けと動的型付けは広く混在している．このバランスが開発者に与える影響についても触れる．型付けモデルは，問題の解決方法を形成し，言語の動作方法を制御する．本書に登場するすべての言語はそれぞれ独自の特徴的な型付けを採用している．
- プログラミングモデルは？ 本書に登場する言語のプログラミングモデルは，オブジェクト指向型，関数型，手続き型，それらのハイブリッド型の 4 つに大別でき

[*1] ［訳注］英語ではそれぞれ strong typing，weak typing，static typing，dynamic typing.

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

る．なかには 2 つ以上のプログラミングモデルを組み合わせた言語もある．本書では，論理ベースのプログラミング言語を 1 つ（Prolog），オブジェクト指向の概念を完全にサポートした言語を 2 つ（Ruby と Scala），関数型の性質を備えた言語を 4 つ（Scala，Erlang，Clojure，Haskell），プロトタイプ言語を 1 つ（Io）取り上げる．なかには Scala のようなマルチパラダイム言語もある．Clojure では，マルチメソッドを用いて独自のパラダイムを実装することさえできる．新しいプログラミングパラダイムを学ぶことは，本書の構想で一番大事なことのひとつだ．

- 言語処理系は？ 言語処理系には，コンパイラ型とインタープリタ型がある．また，仮想マシン上で動作する言語もあれば，そうでない言語もある．本書では，対話型シェルが用意されている言語ではシェルを使っての探索から開始し，より大きなプログラムに取り組むときがきたらファイル単位での処理に移行する．パッケージモデルを本格的に利用するような大規模プロジェクトは扱わない．
- 分岐構文と主要なデータ構造は？ 驚くかもしれないが，`if` や `while` の類を使わずに条件分岐させることができる言語は意外と多い．Erlang のパターンマッチングや Prolog のユニフィケーションがそうだ．コレクションは，ほとんどすべての言語で重要な役割を果たす．Smalltalk や Lisp といった言語では，コレクションによって言語の特性が決まる．C++ や Java といったその他の言語では，コレクションはあちこちで提供されており，凝集度が足りないせいでユーザは不便を強いられる．いずれにしても，コレクションについてはしっかりと理解する必要がある．
- その言語を固有のものにしている主要な機能は？ 並行プログラミングのための高度な機能をサポートする予定がある言語もあれば，Clojure のマクロや Io のメッセージ解釈など，独自の高レベルの構文を提供している言語もある．Erlang の BEAM のように，強力な仮想マシンを用意している言語もある．Erlang では，この仮想マシンのおかげで，他の言語よりもずっと短い期間で耐障害性分散システムを構築できる．また，論理を用いて制約を解決する Prolog のように，特定の問題に特化したプログラミングモデルをサポートしている言語もある．

本書を読み終えると，各言語の専門家にはなれなくても，各言語固有の特性については把握できるようになる．では，本書で扱う言語を紹介しよう．

1.2　この本で扱う言語

本書で紹介する言語を選ぶのは，皆さんが想像するよりずっと簡単だった．単に潜在的な読者に尋ねてみただけだ．結果をまとめてみると，8 つの言語が候補として残った．そのうち JavaScript はあまりにも人気のある言語だったので候補から外し，代わりに，次に人気のあるプロトタイプ言語である Io を選択した．Python も候補から外した．オブジェクト指向言語は 1 つだけで十分だったのと，Ruby が既にリストの上位に入っていたからだ．そこで，リストのトップ 10 に入っていた Prolog が意外な候補として挙がってきた．こうして次の 7 つの言語が残った．これらの言語を選んだ理由を以下に示す．

- *Ruby*．このオブジェクト指向言語は使いやすさとコードの読みやすさという点で

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

評価が高い．オブジェクト指向言語を 1 つも含めないことも少しは考えたが，さまざまなプログラミングパラダイムをオブジェクト指向プログラミング（OOP）と比較したかったので，少なくとも 1 つの OOP 言語を含めることは重要だった．それに筆者は大半のプログラマよりも強く Ruby を推薦したかったし，Ruby の言語デザインを形作ることになった重要な決定事項について，その感じだけでも読者に味わってほしかったということもある．また，Ruby のメタプログラミングについてもかなり突っ込んで説明し，言語のシンタックスの拡張についても触れることができた．結果には大いに満足している．

- *Io*．Prolog と共に，筆者が選んだ言語のなかで最も議論の多い言語だ．商業的には成功していないが，並行性構文とシンプルで統一されたシンタックスは重要な概念だ．余分なものを一切割愛した最小限のシンタックスは強力であり，Lisp と大変よく似ている部分もある．Io は小さなメモリ領域で動作し，JavaScript のようなプロトタイプ言語であり，独特のメッセージディスパッチの仕組みを備えている（これには興味を持ってもらえるのではないかと思う）．
- *Prolog*．確かに古い言語だが，極めて強力な言語でもある．Prolog で数独を解いたことは，筆者にとって目から鱗の経験だった．これまで Java や C で，いくつか難しい問題に苦労させられたが，そうした問題も Prolog なら難なく解けただろう．筆者は，Erlang の作者である Joe Armstrong が助けてくれたおかげで，Prolog をより深く理解することができた（Erlang は Prolog の影響を強く受けている言語だ）．まだ Prolog を使ったことのない方は，この章を読んでうれしい驚きを感じるのではないかと思う．
- *Scala*．Java の仮想マシン（JVM）上で動作する新世代言語の 1 つである Scala は，強力な関数型プログラミングの概念を Java エコシステムにもたらした．と同時に，OOP の概念も全面的に受け入れている．振り返ってみると，C++ は手続き型言語と OOP の橋渡しの役割を担っていた．同じ橋渡し役という意味で，Scala は C++ と驚くほど似ている．Scala のコミュニティに飛び込んでみれば，Scala が純粋な関数型プログラマにとっては異端児だが，Java の開発者には心底喜ばれている理由が分かるだろう．
- *Erlang*．本書で取り上げる言語のなかでも最も古い言語の 1 つだが，並行性，分散処理，耐障害性をきちんとサポートしている関数型言語として最近盛り上がりを見せている．新しく登場したクラウドベースのデータベースである CouchDB の作者は，Erlang を選択し，ひたすら前進してきた．分散処理言語を少しでもかじったことのある人なら，その理由がお分かりだろう．Erlang を使えば，並行分散処理が可能な耐障害性アプリケーションを，これまでの言語では考えられないくらい簡単に設計できる．
- *Clojure*．もう 1 つの JVM 言語である Clojure は，Lisp 方言の 1 つで，JVM 上での並行性の考え方に根本的な変化をもたらすものだ．Clojure は，本書で取り上げた言語のなかでは唯一，バージョン管理データベースと同じ戦略を用いて並行性を管理している言語だ．Lisp 方言である Clojure はすごい威力を備えており，本書で取り上げるプログラミングモデルのなかでも最も高い柔軟性を実現している．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

しかも他の Lisp 方言と違って，括弧の数は大幅に削減されており，膨大な Java ライブラリや広範な配布プラットフォームなど，巨大な Java エコシステムを利用できる．

- *Haskell*．本書で取り上げた言語のなかで唯一の「純粋関数型言語」だ．これは，書き換えることのできる「状態」がどこにも存在しないという意味だ．つまり，同じ関数に同じ入力引数を与えれば，毎回同じ結果が得られる．Haskell は，すべての強く型付けされた言語の中で，最も広範囲で高く評価された型付けモデルをサポートしている．Prolog 同様，理解するには少し時間がかかるが，その価値は十分にある．

お気に入りの言語が含まれていなかったらご容赦願いたい．実は，既に何人かの言語好きの方から非難のメールが届いている．先ほど触れたアンケート調査では数十の言語が候補に含まれていた．筆者が選択した言語は必ずしも最善ではないが，それぞれ独特の個性を備えており，読者の皆さんに何かしら重要なことを教えてくれるはずだ．

1.3　この本を買ってください

もし，あなたが成長したいと思っている有能なプログラマなら，ぜひ本書を買ってほしい．自分が有能なプログラマかどうかなど漠然として分からないかもしれないが，どうかご容赦いただきたい．

学ぶ姿勢を身につける

Dave Thomas は本書の発行元である Pragmatic Bookshelf の創業者の一人だ．彼は，多くの学生達に毎年 1 つの言語を学ぶことを奨励してきた．言語を学ぶことによる効用はいろいろあるが，最低でも，新しく学んだ概念を自分の選んだ言語で書くコードに取り込むようになる．

本書を執筆したことによって，筆者が書く Ruby のコードは大きく変わった．実用性が高くなり，読みやすく，重複が少なくなった．可変変数を使うケースが減り，コードブロックと高階関数によって，コードの質が向上した．Ruby のコミュニティではあまり使われていないテクニックも使うようになり，おかげでコードがコンパクトで読みやすくなった．

うまくいけば，新たなキャリアを開拓することもできる．プログラミングパラダイムは，10 年くらいのサイクルで変化する．Java では窮屈になるばかりなので，筆者は，Web 開発に対する Ruby のアプローチをより深く理解するために，Ruby でさまざまな実験を試みた．いくつか副次的なプロジェクトを成功させた後，自身のキャリアを Ruby による Web 開発の方向に強く押し進め，ひたすら前進してきた．筆者の Ruby のキャリアは基本的な実験から始まって，拡大していったのだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

移行期のヒントとして

本書の読者の多くは，ソフトウェア業界で前回プログラミングパラダイムが切り替わったときのことを覚えていないだろう．オブジェクト指向プログラミングへの移行は出だしで何度かつまずいたが，いずれにしても古い構造型プログラミングパラダイムでは，今日の Web アプリケーションに求められる複雑な処理には対応できなかった．Java が成功したことで，オブジェクト指向への流れが一気に強まり，やがて新しいプログラミングパラダイムとして定着した．これにより，従来のスキルにどっぷりとつかっていた多くの開発者たちは，考え方，使うツール，アプリケーション設計の方法をすべて再編成しなければならなかった．

我々は現在，次のパラダイムシフトの真っ直中にいるのかもしれない．今回のパラダイムシフトを推進しているのはコンピュータの設計の変化だ．本書に登場する 7 つの言語のうち（Ruby と Prolog を除く）5 つの言語が，大変興味深い並行性モデルを採用している．お使いのプログラミング言語がすぐにパラダイムシフトするかどうかはともかく，本書で紹介する言語が非常に興味深い回答を提示してくれることは間違いないだろう．Io のフューチャの実装，Scala のアクター，Erlang の「クラッシュさせろ」の哲学などを確認してみてほしい．Haskell のプログラマがどのような方法で可変状態を使わずに済ませているのか，Clojure がどのようにして，バージョン管理を用いて，最も難しい並行性の問題を解決しているのかを理解してほしい．

意外な箇所で深い理解が得られることもある．例えば，いくつかのクラウドスタイルのデータベースの開発言語として採用されている Erlang は良い例だ．Joe Armstrong 博士は，Prolog を基盤として Erlang の開発を始めた．

1.4　この本を買わないでください

このセクションを読んで同意いただけるまでは本書を買ってはいけない．読者の皆さんには交換条件を提案したい．インストールの詳細ではなくプログラミング言語に焦点を絞って説明することに同意してほしい．代わりに，短い時間でできるだけ多くのことを教えると約束しよう．皆さんには Google で検索してもらう機会が多くなるだろう．また，インストールに関してもサポートを期待してもらっては困る．その分，本題であるプログラミング言語についてより突っ込んだ解説ができるので，本書を読み終えたとき，プログラミング言語に関する皆さんの知識は格段に増えていることだろう．

7 つの言語を解説するというのは，筆者らにとって野心的な試みであることを理解してほしい．読者の皆さんには，脳ミソを最大限に活用して，7 つの異なるシンタックス，4 つのプログラミングパラダイム，40 年間にわたるプログラミング言語技術の蓄積を理解してもらう必要がある．筆者としては，広大な範囲のさまざまなトピックをカバーしなければならない．本書を執筆するために，実際にいくつかの言語を学んだ．各言語の最も重要な部分をうまくカバーするために，簡素化するための前提をいくつか設けることになる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

単なるシンタックスの説明では終わらない

言語設計者の考え方を本当に理解するために，本書では，基本的なシンタックスだけでなく，その背後にある考え方を積極的に吸収してもらう．つまり，お決まりの “Hello, World”，あるいはフィボナッチ数列を出力する程度のコードでは終わらない．Ruby では，メタプログラミングの世界を覗いてみる．Prolog では，完全な数独のパズルを解いてみる．Erlang では，プロセスの死を検出して別のプロセスを起動したり，ユーザに通知したりするモニタを書いてみる．

本書の執筆にあたり，筆者は，基本事項だけにとどまらず，より深い解説をすると決めた．そこで，読者の皆さんに対して，1 つ約束する代わりに，いくつか妥協してもらうことにした．まず，表面を撫でるような説明に甘んじることはしないと約束しよう．その代わり，専門書を読めば見つかるような基本事項には触れない．例外処理についても説明しない（ただし，それが言語の根本に関わる機能の場合は除く）．パッケージモデルを必要としない小さなプロジェクトしか扱わないため，パッケージモデルについても説明しない．さらには，本書で提示するような基本的な問題を解くには必要ないプリミティブ（基本機能）について復習することもしない．

インストールガイドではない

筆者にとって最も大きな問題はプラットフォームだ．筆者はさまざまな書籍の読者から直接問い合わせを受けたことがあり，なかには 3 つの異なる Windows プラットフォーム，OS X，少なくとも 5 つのバージョンの Unix のユーザがいた．それ以外のプラットフォームのさまざまな掲示板にコメントが掲載されているのも見た．7 つの言語を 7 つのプラットフォームについて解説するのは筆者一人ではとても無理だ．共著者がいてもおそらく無理だろう．7 つの言語のインストールについてもサポートは不可能なので，行わない．

読者の皆さんも，内容の古くなったインストールガイドなどに興味はないだろう．言語とプラットフォームは絶えず変化する．言語のインストールガイドのありかと筆者が使っているバージョンについてはそのつど触れる．そうすることで，読者の皆さんが同じ情報源を見て最新の情報を入手して作業できる．というわけで，インストール関連の疑問にはお答えできない．

リファレンスマニュアルではない

本書に掲載したサンプルコードについては，多くの方に徹底してレビューしてもらうように努力した．幸運にも，その言語の設計者本人にレビューしてもらったこともある．レビューが終わる頃には，本書は，各言語の神髄をうまくとらえた教材になるだろうと確信している．とはいうものの，各言語での読者のプログラミング作業のサポートまではできないことをご理解いただきたい．話し言葉にたとえて説明しよう．

通りすがりの旅行者として言葉を覚えても，ネイティブスピーカーとは大きな隔たりがある．筆者は英語は当然流暢に話せるし，たどたどしいがスペイン語も話せる．他にも 3 カ国語で，いくつかのフレーズを知っている．日本で魚を注文したこともあるし，イタリ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:12:43 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

アでトイレの場所を尋ねたこともある．しかし自分の限界は心得ている．プログラミング言語に話を戻すと，筆者は，Basic，C，C++，Java，C#，JavaScript，Ruby，およびその他のいくつかの言語で自由にコーディングできる．その他にも，本書で取り上げた言語も含め，ある程度なら使える言語がいくつかある．本書で取り上げた 7 つの言語のうち 6 つについては，サポートできるだけの知識を持ち合わせていない．Ruby は常時使っており，5 年間の経験があるが，Io で Web サーバを書いたり，Erlang でデータベースアプリケーションを作成する方法を教えることはできない．

7 つの言語の詳細なリファレンスを書こうとしたら，大失敗に終わるだろう．プログラミングガイドを書くとしたら，本書で取り上げた各言語について，短くても本書と同じくらいのボリュームになる．本書では，各言語について，入門者には十分な情報を提供する．各言語をコード例を交えて解説し，プログラム例も掲載する．できる限り，すべてのコードをコンパイルし，確かに動作することを確認するようにする．ただし，読者の皆さんの実際のプログラミング作業を直接サポートすることはできない．

本書で取り上げた言語はすべて，サポートコミュニティが大変充実している．コミュニティが充実していることは，これらの言語を選択した理由の 1 つでもある．各章には，その言語に関する情報を検索してもらう演習問題を用意するようにした．これは，読者の皆さんに自力本願になってほしいという意図があってのことだ．

自分の限界に挑戦してみよう

本書では，20 分のチュートリアルよりは 1 段上の情報を読者の皆さんに提供していく．Google については皆さんご存じだと思うので，7 つの言語すべてについて，簡単な入門者向けの記事を見つけることもできるだろう．本書では対話型のクイックツアーを提供する．また，ちょっとしたプログラミングの課題と 1 つのプログラミングプロジェクトを毎週提示する．これをこなすのは簡単ではないが，有益だし，楽しいはずだ．

本書を単に読むだけでは，シンタックスの雰囲気はつかめるだろうが，それ以上は無理だ．演習問題を自分でコーディングしてみる前にネット上で答えを探すようでは，失敗するだろう．まず自分で解いてみて，いくつかの問題が解けないことを自分自身で認識するようにしてほしい．いつだって，シンタックスを覚えるのは理屈を学ぶより簡単だ．

以上の説明を読んで不安になってしまうようなら，諦めて別の本を探すことをお勧めする．そうした方には満足してもらえないと思うからだ．そのような方は，7 つの言語について，別個のプログラミング解説書を読んだほうがよいだろう．逆に，良いコードを短期間で書けるようにするという目標に興味を持っていただけたなら，ぜひ本書を読んでみてほしい．

1.5 終わりに

ここで，思い切って，やる気を起こさせるような言葉を書くつもりだったが，どうやら，すべては次の言葉に集約されるようだ．

楽しもう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo

第 2 章

Ruby

スプーン一杯の砂糖があるだけで，苦い薬も飲めるのよ．

メリー・ポピンズ

この本を立ち読みしているあなた，あなたと筆者にはどうやら共通点があるようだ．つまり，我々はどちらもプログラミング言語を学習することに興味を持っている．筆者にとって，1 つの言語を学ぶことは，1 人の人物を理解することに似ている．これまでの全キャリアのなかで，筆者は多数の言語を実際に使ってきた．Java は，金持ちの弁護士を兄弟に持つようなものだ．弁護士という人種は若いときは面白いが，歳をとると半径 100 マイル以内のありとあらゆる楽しみをすべて吸い取ってしまうブラックホールと化してしまう．Java もまさにそんな感じだ．Visual Basic は，髪を染めた金髪の美容師のようだった．地球温暖化のような大きな問題は解けないが，髪を切らせたらいつでもそつなくこなすし，話していて面白い．本書では，紹介する各言語を人気のある人物に喩えて説明する．そうすることで，各言語の特徴を少しでも分かりやすく伝えることができればと思っている．

では，筆者のお気に入りの言語の 1 つである Ruby を紹介しよう．Ruby は，ときどきクセのある動作をするが，常に美しく，多少謎めいたところもあり，そして何より魔法のような力を秘めている．イギリスの乳母（ナニー）メリー・ポピンズ[*1]を想像してほしい．当時，乳母といえば C ファミリーの言語のようであった．厳格で，徹底して効率的だが，毎晩飲まなくちゃいけない苦い薬と同じくらい楽しくない．ところがスプーン一杯の砂糖ですべてが変わった．メリー・ポピンズは，家事を楽しいものにし，乳母という仕事から考え得る限りの情熱をうまく引き出すことで，家事という仕事を効率的なものにした．Ruby も，シンタックスシュガー[*2] を，スプーン一杯どころかふんだんに用いて，メリー・ポピンズと同じ変化をもたらした．Ruby の作者である Matz は，言語の効率性には関心がない．彼が関心があるのは，プログラマの生産効率の最適化である．

2.1 簡単な歴史

まつもとゆきひろ氏は 1993 年に Ruby を開発した．たいていの人は彼のことを Matz と呼ぶ．Ruby は，いわゆるスクリプト言語のグループに属し，インタープリタ方式，オブジェクト指向，動的型付けなどの特徴を持つ言語だ．インタープリタ方式というのは，

[*1] 『メリーポピンズ』．DVD．Robert Stevenson 監督作品．1964 年．Walt Disney Video，カリフォルニア州ロサンゼルス 2004 年．

[*2] シンタックスシュガー（構文糖衣）とは，コードの読み書きを容易にする言語の機能を表す言葉である．同じコードを糖衣を外してコアシンタックスで書くこともできる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

Ruby のコードは，コンパイルされるのではなくインタープリタによって実行されるという意味だ．動的型付けというのは，型がコンパイル時ではなく実行時に決まる（バインドされる）という意味だ．一般に，こうした戦略を採用した言語には，柔軟性と実行時の安全性のバランスをとるのが難しいという問題があるのだが，その点については後述する．オブジェクト指向とは，カプセル化（データと振る舞いを 1 つにまとめること），クラスによる継承（クラスツリーによって編成されたオブジェクト型），ポリモーフィズム（多くの形態をとることができるオブジェクト）をサポートしているという意味だ．Ruby は辛抱強く好機を待ち続け，2006 年頃，Rails フレームワークの登場で一気にブレイクした．筆者はエンタープライズ環境で 10 年近くプログラミングに携わっていたが，Ruby のおかげでプログラミングがまた楽しくなってきた．Ruby は実行速度という点ではそれほど効率的ではないが，プログラマの生産性を大幅に高めてくれる．

まつもとゆきひろ氏（Matz）へのインタビュー

筆者は Matz の故郷である松江を訪問し，Ruby 開発の経緯についていろいろと話を聞くことができた．Matz は本書のためにいくつかの質問に快く答えてくれた．

Bruce　Ruby を開発した理由を聞かせてください．

Matz　私は，コンピュータで遊び始めてすぐ，プログラミング言語に興味を持ちました．プログラミング言語はプログラミングの手段であると同時に，プログラミングの考え方を形づくる頭脳を強化してくれる増強装置でもあります．ですから，長い間，趣味として，たくさんのプログラミング言語を学びました．いくつかお遊びで言語を実装したこともありましたが，本物の言語を開発したことはありませんでした．
1993 年に Perl を初めて見たとき，なぜか「Lisp，Smalltalk，Perl の特性を組み合わせたオブジェクト指向言語があれば，生産性を高めてくれる素晴らしい言語になるのでは」というインスピレーションが湧いたのです．それで，そのような言語を開発し始め，Ruby と名付けました．主な動機は自分で楽しむことでした．最初は単なる趣味で，自分の好みに合った言語を作ろうとしたのです．ところがどういうわけか，世界中のプログラマが，Ruby という言語とその考え方に共感してくれたのです．Ruby は私の期待をはるかに超えて，どんどん普及していきました．

Bruce　Ruby のどのような点が最も気に入っていますか？

Matz　Ruby を使うとプログラミングが楽しくなるところが気に入っています．具体的なソフトウェア技術という点では，ブロックが最も気に入っています．ブロックは使いやすくした高階関数のようなものですが，ブロックのおかげで DSL やその他の機能にも大きな可能性が開けました．

Bruce　もし，時間を戻せるとしたら，どの機能を変更したいとお考えですか．

Matz　スレッドを廃止して，アクターやその他の高度な並行性機能を追加したいですね．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

この章を読み進む際には，Ruby を知っているかどうかに関係なく，Matz が設計時にとってきたさまざまなバランスに注目してほしい．シンタックスシュガーを探してみてほしい．シンタックスシュガーとは，言語の基本ルールを崩して，プログラマが少しでも使いやすくなるように，また，コードが少しでも読みやすくなるようにしたものだ．Matz がコードブロックを用いて，コレクションなどで見事な効果を実現している箇所を探してみてほしい．また，Matz がこれまでに実現してきた，シンプルさと安全性，生産性とパフォーマンスの間のバランスを理解するようにしてみてほしい．

では，手始めに，Ruby のコードをいくつか覗いてみよう．

```
>> properties = ['object oriented', 'duck typed', 'productive', 'fun']
=> ["object oriented", "duck typed", "productive", "fun"]
>> properties.each {|property| puts "Ruby is #{property}."}
Ruby is object oriented.
Ruby is duck typed.
Ruby is productive.
Ruby is fun.
=> ["object oriented", "duck typed", "productive", "fun"]
```

Ruby は，筆者に笑顔を取り戻すことを教えてくれた言語だ．とことんダイナミックな Ruby には，素晴らしいサポートコミュニティがある．実装はすべてオープンソースだ．大半の有償サポートは小企業によって提供されている．このため Ruby は，他の言語界に蔓延しているような行きすぎたフレームワークに束縛されずに済んでいる．エンタープライズ分野では出遅れているが，ここへ来て，その高い生産性が，とりわけ Web 開発の分野で効力を発揮している．

2.2 1日目：乳母を探す

メリー・ポピンズは，確かにあらゆる魔法を使うけれども，何よりまず，素晴らしいナニー（子守）である．言語を最初に学ぶときには，（いきなり魔法のような高度な仕事をやらせるのではなく）まずは，その言語を使って，既にやり方が分かっている仕事をやらせてみることだ．そこで，まずは，Ruby と（コンソール経由の）会話形式でやり取りしてみることにしよう．会話はスムーズに流れるだろうか．それとも，どうにもぎこちないものになるだろうか．Ruby では，どのようなプログラミングモデルを採用しているのだろうか．型はどのように扱うのだろうか．これらの疑問に答えていくことにしよう．

クイックツアー

前にお約束したとおり，本書ではすぐに古くなるインストール手順を詳しく説明することはしない．第一，Ruby ではその必要もない．Ruby のインストールは実に簡単だからだ．`http://www.ruby-lang.org/ja/downloads/` に移動して，お使いのプラットフォームを選択し，Ruby 1.8.6 以降をインストールすればよい．Windows をお使いなら，ワンクリック・インストーラーが用意されている．OS X Leopard 以降をお使いなら，Xcode インストール用ディスクを使えば Ruby がインストールされる．

正しくインストールされたかどうかを確認するには，`irb` と入力すればよい．エラーが表示されなければ，準備完了だ．この章を読み進めてほしい．エラーが出ても大丈夫．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

インストール時に発生する問題はたいてい他の誰かが既に解決済みだ．Google で検索してみれば，解決方法が分かるだろう．

Ruby をコンソールから使う

まず，irb と入力してみてほしい．Ruby の対話型コンソールが表示される．何かコマンドを入力すると，応答が返ってくる．試しに次のコマンドを入力してみてほしい．

```
>> puts 'hello, world'
hello, world
=> nil
>> language = 'Ruby'
=> "Ruby"
>> puts "hello, #{language}"
hello, Ruby
=> nil
>> language = 'my Ruby'
=> "my Ruby"
>> puts "hello, #{language}"
hello, my Ruby
=> nil
```

Ruby について知らない読者でも，上の簡単なやり取りは，Ruby について多くのことを教えてくれる．Ruby がインタープリタ型言語であることはご存じだろう．実際，Ruby はほとんどの場合，インタープリタ方式で動作する（ただし，実行時に Ruby をバイトコードにコンパイルする仮想マシンの開発に取り組んでいる開発者もいる）．上の例では，変数は一切宣言していない．何か実行すると，戻り値を返すように指定していなくても必ず値が返ってくる．Ruby では，コードが何らかの値を返す．

上の例には，少なくとも 2 種類の文字列が登場する．一重引用符で囲まれた文字列はそのまま解釈され，二重引用符で囲まれた文字列は評価される．Ruby インタープリタによって評価されるモノの 1 つに，文字列置換がある．上の例では，コード language によって返された値を文字列に埋め込んでいる．では次に進もう．

プログラミングモデル

プログラミング言語に関する質問のうち，最初に答えるべきは「どのようなプログラミングモデルを採用しているか」という質問だ．この答えは必ずしも簡単ではない．読者のなかには C，Fortran，Pascal などの手続き型言語を経験したことのある人も多いだろう．今ではほとんどの開発者がオブジェクト指向言語を使っているが，大半のオブジェクト指向言語が手続き的な要素も持ち合わせている．例えば Java では，整数の 4 はオブジェクトではない．読者のなかには，関数型プログラミング言語について調べるために本書を選んだ人もいるかもしれない．Scala などの関数型プログラミング言語では，オブジェクト指向の概念を取り入れることによって複数のプログラミングモデルが混在している．PostScript や Forth などのスタック型プログラミング言語では，言語の中心的な機能としてスタックを使う．Io，Lua，Self などのプロトタイプ言語では，オブジェクトの定義だけでなく継承の実現にも，クラスではなくオブジェクトを使う．

Ruby は純粋なオブジェクト指向言語だ．この章では，Ruby の徹底したオブジェクト指向ぶりについて見ていく．まず，いくつかの基本的なオブジェクトを見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
>> 4
=> 4
>> 4.class
=> Fixnum
>> 4 + 4
=> 8
>> 4.methods
=> ["inspect", "%", "<<", "singleton_method_added", "numerator", ...
 "*", "+", "to_i", "methods", ...
 ]
```

最後のメソッドのリストは紙面の都合で一部しか表示していないが，感じは分かると思う．Rubyでは，個々の数値にいたるまで，あらゆるものがオブジェクトだ．数値はFixnumというクラスを持つオブジェクトだ．methodsメソッドはメソッドの配列を返す（Rubyでは配列を角括弧で表す）．このようにドット演算子（.）を使って，オブジェクトのあらゆるメソッドを呼び出すことができる．

判定

プログラムの存在目的は判断を下すことである．したがって，プログラム言語が判断を下す方法が，その言語でコードを書き，その言語で考える方法を形作る中心的な概念となるのも当然だ．Rubyは，多くの点で，大半のオブジェクト指向言語および手続き型言語と似ている点がある．次の式を見てほしい．

```
>> x = 4
=> 4
>> x < 5
=> true
>> x <= 4
=> true
>> x > 4
=> false
>> false.class
=> FalseClass
>> true.class
=> TrueClass
```

Rubyには，trueまたはfalseに評価される式がある．ご推察どおり，trueとfalseもファーストクラス（第一級）[*3]オブジェクトだ．これらを使ってコードを条件分岐させることができる．

```
>> x = 4
=> 4
>> puts 'This appears to be false.' unless x == 4
=> nil
>> puts 'This appears to be true.' if x == 4
This appears to be true.
=> nil
>> if x == 4
>>   puts 'This appears to be true.'
>> end
```

[*3] ［訳注］引数や戻り値として扱うことができるデータという意味．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
This appears to be true.
=> nil
>> unless x == 4
>>   puts 'This appears to be false.'
>> else
?>   puts 'This appears to be true.'
>> end
This appears to be true.
=> nil
>> puts 'This appears to be true.' if not true
=> nil
>> puts 'This appears to be true.' if !true
=> nil
```

筆者は，簡単な条件文を採用した Ruby の設計が大変気に入っている．if 文または unless 文には，ブロック形式（if 条件, 文, end）と 1 行形式（文 if 条件）がある．

1 行形式の if 文については当惑する人もいるようだが，1 つの思考の道筋（アイデア）を 1 行のコードで表現できるので，筆者は便利だと思っている．

```
order.calculate_tax unless order.nil?
```

もちろん，上のコードはブロック形式でも書けるが，単一の明瞭な思考の道筋に余計なノイズを追加してしまうことになる．簡単なアイデアを 1 行にまとめることができれば，コードを読むのが楽になる．unless についても同じ理由で気に入っている．unless で書けることは not または!を使って書けるが，unless を使ったほうがずっと読みやすくなる．

while と until はほぼ同じだ．

```
>> x = x + 1 while x < 10
=> nil
>> x
=> 10
>> x = x - 1 until x == 1
=> nil
>> x
=> 1
>> while x < 10
>>   x = x + 1
>>   puts x
>> end
2
3
4
5
6
7
8
9
10
=> nil
```

=は代入，==は等価性の判定である点に注意してほしい．Ruby では，オブジェクトごとに等価性の概念が定義される．例えば，数値は値が等しいときに等価になる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

trueとfalse以外の値を式として使うこともできる．

```
>> puts 'This appears to be true.' if 1
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 'random string'
(irb):31: warning: string literal in condition
This appears to be true.
=> nil
>> puts 'This appears to be true.' if 0
This appears to be true.
=> nil
>> puts 'This appears to be true.' if true
This appears to be true.
=> nil
>> puts 'This appears to be true.' if false
=> nil
>> puts 'This appears to be true.' if nil
=> nil
```

上の例からお分かりのように，nilとfalse以外のものはすべてtrueに評価される．0はtrueなので，CおよびC++のプログラマは注意してほしい．

論理演算子の動作は，細かい点でいくつか違いはあるが，C，C++，C#，Javaの論理演算子とほぼ同じである．and（または&&）は論理積，or（または||）は論理和だ．これらの演算子を使った判定では，判定結果が明らかになった時点でインタープリタは実行を中止する．式全体を実行して比較する場合は，&または|を使うこと．以下に例を示す．

```
>> true and false
=> false
>> true or false
=> true
>> false && false
=> false

>> true && this_will_cause_an_error
NameError: undefined local variable or method `this_will_cause_an_error'
    for main:Object
        from (irb):59
>> false && this_will_not_cause_an_error
=> false
>> true or this_will_not_cause_an_error
=> true
>> true || this_will_not_cause_an_error
=> true
>> true | this_will_cause_an_error
NameError: undefined local variable or method `this_will_cause_an_error'
    for main:Object
        from (irb):2
        from :0
>> true | false
=> true
```

以上，ここまでは何のからくりもない．これらのコマンドには通常，短絡評価バージョン（||や&&）を使う．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ダックタイピング

次に，Ruby の型モデルについて簡単に見ていこう．まず知っておく必要があるのは，プログラマが型を間違えたとき，Ruby ではどの程度，安全性が保護されるのかという点だ．つまり，型安全性の問題だ．強く型付けされた言語では，特定の演算について型をチェックし，実行時に問題が起こらないように事前に型をチェックする．このチェックを行うタイミングとして，コードをインタープリタまたはコンパイラに渡すとき，またはコードを実行するときの 2 つがある．次のコードを見てほしい．

```
>> 4 + 'four'
TypeError: String can't be coerced into Fixnum
        from (irb):51:in `+'
        from (irb):51

>> 4.class
=> Fixnum
>> (4.0).class
=> Float

>> 4 + 4.0
=> 8.0
```

このように，Ruby は強く型付けされている[*4]．つまり，型が衝突するとエラーになる．Ruby はこの型チェックをコンパイル時ではなく実行時に行う．この点を証明するために，少し早いが，関数を定義する方法を示す．キーワード def は，関数を定義するが，実行はしない．次のコードを入力してみてほしい．

```
>> def add_them_up
>>   4 + 'four'
>> end
=> nil
>> add_them_up
TypeError: String can't be coerced into Fixnum
        from (irb):56:in `+'
        from (irb):56:in `add_them_up'
        from (irb):58
```

見てのとおり，Ruby は，実際にコードを実行するまで型チェックを実行しない．この概念を動的型付けと呼ぶ．ただし，動的型付けにも欠点がないわけではない．動的型付けでは，静的型付けを採用した場合に比べて捕捉できるエラーの数が少なくなる．というのは，静的に型付けされたシステムを使った場合のほうが，コンパイラとツールによってより多くのエラーを捕捉できるからだ．しかし Ruby の型システムには潜在的な利点もある．それは，クラスは，同じように使うからといって，同じ親から（メソッドを）継承する必要はないというものだ．

```
>> i = 0
```

[*4] ここで筆者は少し嘘をついている．といってもほんの少しだが．この章の 3 日目の例では，実行時に既存のクラスを変更して見せている．論理的には，ユーザはクラスを原型をとどめないくらいに変更し，型安全性を壊してしまうこともできるので，厳密な意味では，Ruby は強く型付けされているとはいえない．しかしほとんどの場合，強く型付けされた言語と同じように振る舞う．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:28:55 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
=> 0
>> a = ['100', 100.0]
=> ['100', 100.0]
>> while i < 2
>>   puts a[i].to_i
>>   i = i + 1
>> end
100
100
```

このコードはダックタイピングの実例だ．この配列の最初の要素は String，2 番目の要素は Float，と型が異なるのだが，同じコード（つまりループ本体）が，to_i によって各要素を整数に変換している．ダックタイピングでは，基盤となる型が何であっても構わない．アヒルのように歩き，アヒルのようにガーガー鳴く（quack）動物がいたら，それはアヒルなのである．この例では，quack メソッド（アヒルのように鳴くメソッド）に相当するのは，to_i メソッドである．

ダックタイピングは，純粋なオブジェクト指向設計を行う場合には極めて重要となる．オブジェクト指向の設計哲学における重要な教義は，実装ではなく，インタフェースに合わせてコーディングするというものだ．ダックタイピングを導入すると，ほとんど何の作法もなしに，この設計哲学を簡単に適用できる．push メソッドと pop メソッドが使えるオブジェクトは，スタックとして扱うことができる．そうでなければ，スタックとしては扱えない．

1 日目で学んだこと

ここまで，基本事項についてざっと説明してきた．Ruby はインタープリタ方式のオブジェクト指向言語だ．すべてのモノはオブジェクトであり，メソッドやクラスなど，オブジェクトの部品にアクセスすることも簡単にできる．Ruby はダックタイプ型言語であり，強い型付けの言語とほぼ同じように動作するが，両者の違いについては一部の専門家の間で議論がある．Ruby は，NilClass や String といったコアクラスの変更を含め，何でもできる自由奔放な言語である．では，いよいよ，セルフスタディを始めてもらうことにしよう．

セルフスタディ 1 日目

さて，Ruby と初めてのデートをしてもらったところで，そろそろちょっとしたコードを書いてみることにしよう．このセッションでは，完全なプログラムを書くのではなく，irb を使って小さな Ruby のスクリプトを実行する．全力で取り組んでみたうえで次に進んでほしい．

探してみよう：

- Ruby API
- *Programming Ruby: The Pragmatic Programmer's Guide* [TFH08] のオンライン版
- 文字列の一部を置換するメソッド
- Ruby の正規表現に関する情報

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- Ruby の `Range` に関する情報

試してみよう：

- 文字列 “Hello, world” を出力する．
- 文字列 “Hello, Ruby” の中の “Ruby” という単語のインデックスを検索する．
- 自分の名前を 10 回出力する．
- 文字列 “This is sentence number 1” の 1 を 10 までカウントアップしながら 10 回出力する．
- ファイルに格納されている Ruby のプログラムを実行する．
- ボーナス問題：少し物足りない人は，乱数を選択するプログラムを書いてみてほしい．プレーヤーに数字を選択してもらい，その数字が生成された乱数よりも大きいか小さいかを返す．
  （ヒント：`rand(10)` は 0〜9 までの乱数を生成する．`gets` はキーボードから文字列を読み込む．この文字列をプログラムで整数に変換する．）

2.3　2 日目：空からふわふわ舞い降りる

映画『メリーポピンズ』で公開当時最も画期的だったシーンの 1 つは，彼女の登場シーンだった．傘で風に乗って街にふわふわと舞い降りたのだ．筆者の子供たちは，あの登場シーンがなぜ画期的だったのか決して分からないだろう．2 日目の今日は，Ruby を動かしているちょっとしたマジックを体験してもらおう．具体的には，オブジェクト，コレクション，クラスなどの基本構成ブロックの使い方について学習する．コードブロックの基礎についても学習する．既存の考え方にとらわれず，心を開いて Ruby のマジックを受け入れてほしい．

関数の定義

Java や C#と違って，関数を定義するためにわざわざクラス全体を構築する必要はない．次のようにコンソールから直接関数を定義できる．

```
>> def tell_the_truth
>>   true
>> end
```

すべての関数は戻り値として何かを返す．明示的に戻り値を指定しないと，関数は，終了前に実行された最後の式の値を返す．他のすべてのものと同様，この関数もオブジェクトである．後で，他の関数の引数として関数を渡す方法について説明する．

配列

配列は，順序付きコレクションであり，Ruby の中心となる極めて有用な機能である．Ruby 1.9 で順序付きハッシュも導入されたが，一般に Ruby で順序付きコレクションといえば配列だ．次の例を見てほしい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
>> animals = ['lions', 'tigers', 'bears']
=> ["lions", "tigers", "bears"]
>> puts animals
lions
tigers
bears
=> nil
>> animals[0]
=> "lions"
>> animals[2]
=> "bears"
>> animals[10]
=> nil
>> animals[-1]
=> "bears"
>> animals[-2]
=> "tigers"
>> animals[0..1]
=> ['lions', 'tigers']
>> (0..1).class
=> Range
```

見てのとおり，Ruby のコレクションは融通が利く．例えば，未定義の配列要素にアクセスすると nil が返される．また，配列としての機能強化ではなく，使いやすさを高める機能もいくつか用意されている．animals[-1] は配列の最後の要素を，animals[-2] は最後から 2 番目の要素を返す．これらの機能はシンタックスシュガーと呼ばれる．つまり，使いやすさを高めるために追加された機能だ．animals[0..1] という式はシンタックスシュガーのように見えるかもしれないが，そうではない．0..1 は実は Range であり，0〜1 まで（境界を含む）のすべての数を意味する．

配列は他の型を保持することもできる．

```
>> a[0] = 0
NameError: undefined local variable or method `a' for main:Object
from (irb):23
>> a = []
=> []
```

おっと．配列になる前の配列を使おうとしてしまった．このエラーは，Ruby の配列とハッシュの動作方法に関するヒントを与えてくれる．[] は，実は Array クラスのメソッドだ．

```
>> [1].class
=> Array
>> [1].methods.include?('[]')
=> true
>> # use [1].methods.include?(:[]) on ruby 1.9
```

つまり [] と []=は，配列にアクセスするためのシンタックスシュガーなのだ．これを正しく使うには，まず空の配列を作成する．あとは，次のように，作成した配列を自由に操作できる．

```
>> a[0] = 'zero'
=> "zero"
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
>> a[1] = 1
=> 1
>> a[2] = ['two', 'things']
=> ["two", "things"]
>> a
=> ["zero", 1, ["two", "things"]]
```

このように配列に格納する値の型は異なっていても構わない．

```
>> a = [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
=> [[1, 2, 3], [10, 20, 30], [40, 50, 60]]
>> a[0][0]
=> 1
>> a[1][2]
=> 30
```

配列の配列である多次元配列も使える．

```
>> a = [1]
=> [1]
>> a.push(1)
=> [1, 1]
>> a = [1]
=> [1]
>> a.push(2)
=> [1, 2]
>> a.pop
=> 2
>> a.pop
=> 1
```

配列には豊富な API が用意されており，まるでキュー，リスト，スタック，セットなどのように使うことができる．次に，Ruby のもう 1 つの主要なコレクションであるハッシュについて見ていこう．

ハッシュ

コレクションはオブジェクトの入れ物であることを思い出してほしい．ハッシュという入れ物には，すべてのオブジェクトにラベルが付く．ラベルがキーで，オブジェクトが値となる．ハッシュは，キーと値のペアの集まりである．

```
>> numbers = {1 => 'one', 2 => 'two'}
=> {1=>"one", 2=>"two"}
>> numbers[1]
=> "one"
>> numbers[2]
=> "two"
>> stuff = {:array => [1, 2, 3], :string => 'Hi, mom!'}
=> {:array=>[1, 2, 3], :string=>"Hi, mom!"}
>> stuff[:string]
=> "Hi, mom!"
```

これはさほど難しくない．ハッシュの動作は配列とほぼ同じだが，ハッシュでは，整数のインデックスの代わりに，任意のキーを使うことができる．上の例の最後に示したハッ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

シュは面白い．というのは，本書で初めてシンボルが登場しているからだ．シンボルは，:symbol のように前にコロンが付いた識別子だ．シンボルは，モノやアイデアに名前を付けるとき便利だ．同じ値を持つ 2 つの文字列は物理的に異なる文字列である可能性があるが，同じシンボルは物理的に同じオブジェクトだ．このことは，次のように同じシンボルのオブジェクト識別子を数回取得してみれば分かる．

```
>> 'string'.object_id
=> 3092010
>> 'string'.object_id
=> 3089690
>> :string.object_id
=> 69618
>> :string.object_id
=> 69618
```

ハッシュはときどき普通とは少し違った場面で使われる．例えば Ruby では名前付き引数をサポートしていないが，これはハッシュでシミュレートすることができる．ちょっとしたシンタックスシュガーを追加すれば，次のように面白い動作をさせることができる．

```
>> def tell_the_truth(options={})
>>   if options[:profession] == :lawyer
>>     'it could be believed that this is almost certainly not false.'
>>   else
>>     true
>>   end
>> end
=> nil
>> tell_the_truth
=> true
>> tell_the_truth :profession => :lawyer
=> "it could be believed that this is almost certainly not false."
```

このメソッドは，省略可能な単一の引数 options をとる．引数を省略すると，options には空のハッシュが設定される．キーとして:profession，値として:lawyer を渡すと，文字列が返される（まるで弁護士が言いそうな，あいまいな文言だ）．しかし内容が何であれ，文字列は真として評価されるので，true が返されたのと変わらない．引数をブレースで囲んでいない点に注意してほしい．関数の最後の引数を囲むブレースは省略可能だ．配列要素，ハッシュキー，ハッシュ値には，ほとんど何でも保持できるため，Ruby では驚くほど高度なデータ構造体を作れる．ただ，Ruby が本当の力を発揮するのは，コードブロックを使い始めたときだ．

コードブロックと yield

コードブロックとは名前のない関数だ．コードブロックは，関数やメソッドに引数として渡すことができる．次に例を示す．

```
>> 3.times {puts 'hiya there, kiddo'}
hiya there, kiddo
hiya there, kiddo
hiya there, kiddo
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ブレースで囲まれたコードは，コードブロックと呼ばれる．times は Fixnum のメソッドで，何かの処理（コードブロック）を指定された回数（Fixnum 値）だけ繰り返す．コードブロックは，{/}または do/end で囲んで指定する．Ruby の慣習では，1 行だけのコードブロックはブレースで囲み，複数行にまたがるコードブロックは do/end で囲む．コードブロックには 1 つまたは複数の引数を指定できる．

```
>> animals = ['lions and ', 'tigers and', 'bears', 'oh my']
=> ["lions and ", "tigers and", "bears", "oh my"]
>> animals.each {|a| puts a}
lions and
tigers and
bears
oh my
```

上のコードは，コードブロックの威力の片鱗を見せている．このコードは，コレクション内の各アイテムに対して実行する処理を Ruby に指示している．わずかなコードを書くだけで，Ruby は各要素に対して繰り返し，その内容を出力する．何が起こっているのか感じをつかんでもらうために，カスタムの times メソッドを書いてみよう．

```
>> class Fixnum
>>   def my_times
>>     i = self
>>     while i > 0
>>       i = i - 1
>>       yield
>>     end
>>   end
>> end
=> nil
>> 3.my_times {puts 'mangy moose'}
mangy moose
mangy moose
mangy moose
```

上のコードでは，既存のクラスを開いてメソッドを追加している．この例では，my_times というメソッドが一定回数のループを実行し，ループ内で yield を用いてコードブロックを呼び出している．ブロックはメソッドの引数にすることもできる．次の例を見てほしい．

```
>> def call_block(&block)
>>   block.call
>> end
=> nil
>> def pass_block(&block)
>>   call_block(&block)
>> end
=> nil
>> pass_block {puts 'Hello, block'}
Hello, block
```

このテクニックを使うと，実行可能なコードを受け渡しできる．ブロックの用途は何も繰り返しだけではない．Ruby ではブロックを用いてさまざまな処理を実行できる．以

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

下にいくつか例を挙げる．

実行を遅延させる：

```
execute_at_noon { puts 'Beep beep... time to get up'}
```

条件分岐を実現する：

```
... 何かコード...
in_case_of_emergency do
  use_credit_card
  panic
end

def in_case_of_emergency
  yield if emergency?
end
... さらにコード...
```

ポリシーを実施する：

```
within_a_transaction do
  things_that
  must_happen_together
end

def within_a_transaction
  begin_transaction
  yield
  end_transaction
end
```

ほかにも多くの用途がある．例えば，ファイルの各行を処理する，HTTP トランザクション内で処理を実行する，コレクションに対して複雑な演算を実行する，といった処理をブロックを使って実装している Ruby のライブラリがある．Ruby では，いたるところでブロックが使われる．

ファイルから Ruby を実行する

サンプルコードが少し複雑になってきたので，対話型コンソールから作業するのは不便になってきた．ちょっとしたコードを書くときはコンソールを使えばよいが，たいていはコードをファイルに格納する．hello.rb というファイルを作成することにしよう．このファイルには任意の Ruby のコードを格納できる．

```
puts 'hello, world'
```

このファイルを現在のディレクトリに保存し，コマンドラインから実行する．

```
batate$ ruby hello.rb
hello, world
```

Ruby を本格的な統合開発環境で使っているユーザもいるが，大半のユーザは簡単なエディタでファイルを編集する方法で満足している．筆者のお気に入りのエディタは TextMate だが，vi，Emacs，その他多数の人気のあるエディタで Ruby 用のプラグインが用意されている．この点を理解したうえで，Ruby プログラムの再利用可能な構築ブロッ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

クの説明に進むことにしよう．

クラスの定義

Java，C#，C++ と同様，Ruby にもクラスとオブジェクトがある．クッキーの抜き型とクッキーを思い浮かべてほしい．クラスとはオブジェクトのひな形のことである．もちろん，Ruby では継承もサポートしている．ただし C++ と違って，Ruby のクラスは 1 つの親（スーパークラス）からしか継承できない．実際にそうなっていることをコンソールから確認するために，次のように入力してみてほしい．

```
>> 4.class
=> Fixnum
>> 4.class.superclass
=> Integer
>> 4.class.superclass.superclass
=> Numeric
>> 4.class.superclass.superclass.superclass
=> Object
>> 4.class.superclass.superclass.superclass.superclass
=> nil
```

ここまでは予想どおりだ．オブジェクトはクラスから導出される．4 のクラスは Fixnum であり，Fixnum は，Integer，Numeric，そして最後に Object から継承している[*5]．

図 2.1「Ruby のメタモデル」(次ページ)で，全体の様子を確認してほしい．継承元をたどっていくと，すべてのクラスは最終的には Object に行き着く．Class は Module でもある．Class のインスタンスはオブジェクトのテンプレートとなる．上の例では，Fixnum は Class のインスタンスであり，4 は Fixnum のインスタンスである．これらの各クラスも実はオブジェクトでもある．

```
>> 4.class.class
=> Class
>> 4.class.class.superclass
=> Module
>> 4.class.class.superclass.superclass
=> Object
```

したがって，Fixnum は Class クラスから導出される．ここからが少し混乱するところだ．Class は Module から継承しており，Module は Object から継承している．このようにクラス階層を上向きにたどっていくと，Ruby のすべてのクラスは，共通の祖先 Object にたどり着く．

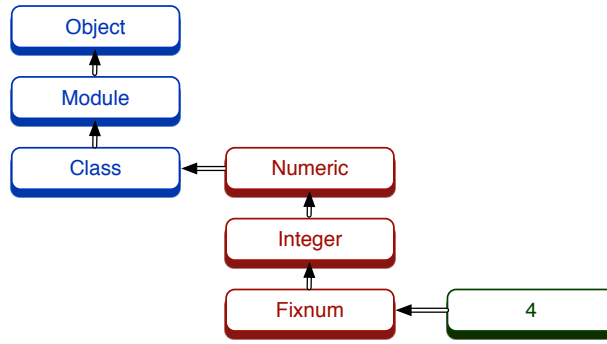
▶ code/ruby/tree.rb

```
class Tree
  attr_accessor :children, :node_name
```

[*5] ［訳注］Ruby 1.9 以降では，Object クラスのスーパークラスとして（最上位の）BasicObject が導入されている．このため，実行例の最後の行では nil の代わりに BasicObject が返される．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲ 図 2.1　Ruby のメタモデル

```
  def initialize(name, children=[])
    @children = children
    @node_name = name
  end

  def visit_all(&block)
    visit &block
    children.each {|c| c.visit_all &block}
  end

  def visit(&block)
    block.call self
  end
end

ruby_tree = Tree.new( "Ruby",
  [Tree.new("Reia"),
   Tree.new("MacRuby")] )

puts "Visiting a node"
ruby_tree.visit {|node| puts node.node_name}
puts

puts "visiting entire tree"
ruby_tree.visit_all {|node| puts node.node_name}
```

この多機能で強力なクラスは，極めてシンプルなツリーを実装している．initialize，visit，visit_all の 3 つのメソッドと，children と node_name の 2 つのインスタンス変数が定義されている．initialize には特別な意味がある．すなわち，クラスが新しいオブジェクトをインスタンス化するとき，Ruby によって呼び出されるのが initialize だ．

ここで Ruby の慣例と規則をいくつか紹介しておく．クラスは大文字で始まり，通常，キャメルケースを使う．インスタンス変数（オブジェクトごとに 1 つの値を持つ）の

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

先頭には@を，クラス変数（クラスごとに 1 つの値を持つ）の先頭には@@を付ける必要がある．インスタンス変数とメソッド名は，小文字で始まり，区切りにアンダースコアを使う（underscore_style）．定数はすべて大文字で書く（ALL_CAPS）．上のコードでは，ツリークラスを定義している．各ツリーには，2 つのインスタンス変数@children と @node_name がある．判定用の関数とメソッドには通常，疑問符を付ける（if test?）．

attr キーワードはインスタンス変数を定義する．これにはいくつか種類がある．最もよく使われるのは，attr（インスタンス変数およびその変数にアクセスするための同名のメソッドを定義）と attr_accessor（インスタンス変数，アクセサ，セッターを定義）である．

上のプログラムは密度が高く，すごい威力を備えている．まず，ブロックと再帰を用いて，任意のユーザがツリー内のすべてのノードを訪問できるようにしている．Tree の各インスタンスは，ツリーの 1 ノードを保持している．initialize メソッドは，children と node_name の初期値を与える．visit メソッドは，引数として受け取ったコードブロックを呼び出す．visit_all メソッドは，当該ノードの visit を呼び出し，そのすべての子について再帰的に visit_all を呼び出す．

残りのコードでは準備したメソッドを使っている．まずツリーを定義し，1 つのノードを訪問し，その後すべてのノードを訪問している．これを実行すると次のような出力が得られる．

```
Visiting a node
Ruby

visiting entire tree
Ruby
Reia
MacRuby
```

クラスだけではまだ一方を知ったに過ぎない．実は先のコード（p. 24）に簡単なモジュールが登場していた．戻って詳しく見てみよう．

Mixin を書く

オブジェクト指向言語では，継承の仕組みを使って，類似のオブジェクトに振る舞いを伝搬する．振る舞いが異なる場合は，複数のクラスから継承するか（多重継承），他の解決策を考える．これまでの経験から，多重継承は複雑で問題が多いことが分かっている．Java ではインタフェースを使ってこの問題を解決している．Ruby ではモジュールを使う．モジュールとは関数と定数の集まりである．クラスの一部としてモジュールを混ぜ込むと，混ぜ込まれた振る舞いと定数が，そのクラスの一部になる．

例えば，任意のクラスに to_f メソッドを追加する次のようなクラスを考えてみる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:33:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/ruby/to_file.rb

```
module ToFile
  def filename
    "object_#{self.object_id}.txt"
  end

  def to_f
    File.open(filename, 'w') {|f| f.write(to_s)}
  end
end

class Person
  include ToFile
  attr_accessor :name

  def initialize(name)
    @name = name
  end

  def to_s
    name
  end
end

Person.new('matz').to_f
```

まずモジュールの定義から始めよう．このモジュールには 2 つのメソッドがある．to_f メソッドは，to_s メソッドの出力を，filename メソッドによって提供されたファイル名を持つファイルに書き出す．ここで面白いのは，to_s はモジュールで使われているが，クラスで実装されている点だ．しかも，モジュールで to_s が使われている時点では，クラスは定義さえされていない．モジュールは，include 元のクラスと奥底のレベルでやり取りする．モジュールは複数のクラスメソッドに依存していることが多い．Java では，このようなやり取りのための取り決め（契約）を明示的に指定する．すなわち，クラスが正式なインタフェースを実装する．Ruby では，この取り決めをダックタイピングによって暗黙に行う．

Person クラスの実装には特に面白いことは何もない．が，そこがポイントでもある．Person に ToFile モジュールを include すればそれで終わり．ファイルに書き出す機能は，実際のクラスが Person であるかどうかとはまったく関係ない．そこで，中身をファイルに書き出す機能を，その機能を「混ぜ込む」（mixin）ことによって追加している．Person に新しい mixin やサブクラスを追加することもできる．各サブクラスは，追加されたすべての mixin の機能を，その実装を意識することなく使うことができる．つまり，単純な単一継承を使用してクラスの中核部分を定義し，そこにモジュールで機能を追加できる．このプログラミングスタイルは *mixin* と呼ばれ，Flavors で初めて導入され，Smalltalk や Python など多くの言語で使われている．mixin を実現している仕組みが常にモジュールと呼ばれるわけではないが，はっきりしているのは，単一継承と mixin の組み合わせは優れた動作を実現するということだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

モジュール，Enumerable，セット

Ruby で最も重要な mixin として，Enumerable と Comparable がある．Enumerable を include するクラスでは each を，Comparable を include するクラスでは<=>を実装しておく必要がある．その形状から宇宙船演算子と呼ばれる<=>は，2 つの値を a <=> b のように比較し，「a が b よりも小さい」「a が b よりも大きい」「2 つの項が等しい」の 3 つのケースに応じてそれぞれ-1，1，0 を返す．Enumerable と Comparable を使えば，下記のようなコレクションを操作する多数の便利なメソッドが，クラスでいちいち実装しなくても使えるようになる．コンソールを開いて確認してみよう．

```
>>  'begin' <=> 'end'
=> -1
>> 'same' <=> 'same'
=> 0
>> a = [5, 3, 4, 1]
=> [5, 3, 4, 1]
>> a.sort
=> [1, 3, 4, 5]
>> a.any? {|i| i > 6}
=> false
>> a.any? {|i| i > 4}
=> true
>> a.all? {|i| i > 4}
=> false
>> a.all? {|i| i > 0}
=> true
>> a.collect {|i| i * 2}
=> [10, 6, 8, 2]
>> a.select {|i| i % 2 == 0 } # even
=> [4]
>> a.select {|i| i % 2 == 1 } # odd
=> [5, 3, 1]
>> a.max
=> 5
>> a.member?(2)
=> false
```

any?は，コレクション内のいずれかの要素について，指定した条件が真であれば true を返す．all?は，コレクション内のすべての要素について，指定した条件が真であれば true を返す．こうした整数比較を行う宇宙船演算子は Fixnum で実装されているため，ソートして最小値や最大値を求めることができる．

集合演算も実行できる．collect と map は，コレクションの各要素に関数を適用して，結果として生成された配列を返す．find は，指定された条件に一致する要素を 1 つ見つける．select と find_all は，条件に一致するすべての要素を返す．また，次のように inject を使用して，リストのすべての要素の和や積を求めることもできる．

```
>> a
=> [5, 3, 4, 1]
>> a.inject(0) {|sum, i| sum + i}
=> 13
>> a.inject {|sum, i| sum + i}
=> 13
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
>> a.inject {|product, i| product * i}
=> 60
```

inject は少し難しく感じるが，実はそうでもない．inject メソッドは，2 つの引数と 1 つの式からなるコードブロックを受け取る．このコードブロックはリスト内の各項目について実行され，その際，リストの各要素がコードブロックの第 2 引数として渡される．第 1 引数は，コードブロックの前回の実行結果だ．コードブロックの初回の実行時には前回の実行結果がないため，inject の引数として初期値を指定する（この引数を省略すると，コレクションの最初の要素と 2 番目の要素がブロックに渡される）．今度は，累算の推移が分かるようにコメント付きで出力したものを見てみよう．

```
>> a.inject(0) do |sum, i|
?>  puts "sum: #{sum}  i: #{i}   sum + i: #{sum + i}"
?>  sum + i
?>end
sum: 0  i: 5   sum + i: 5
sum: 5  i: 3   sum + i: 8
sum: 8  i: 4   sum + i: 12
sum: 12  i: 1   sum + i: 13
```

期待どおり，直前の行の結果が常に，次の行の最初の値になっている．inject を使うと，多数の文章の単語数を計算したり，段落中で最も長い単語を見つけたりすることができる．

2 日目で学んだこと

2 日目では，Ruby の利点やちょっとしたマジックを紹介した．Ruby の柔軟性が見えてきたところだと思う．コレクションはいたってシンプルだ．2 つのコレクション（配列とハッシュ）に複数の API が定義されている．アプリケーションのパフォーマンスは二の次である．Ruby にとって重要なのは，プログラマのパフォーマンスだ．Enumerable モジュールを見れば，Ruby の設計の良さの片鱗をうかがい知ることができる．単一継承のオブジェクト指向言語は決して珍しくはないが，Ruby の実装には，直感的な設計と使いやすい機能が凝縮されている．このレベルの抽象化でも少しは優れたプログラミング言語になるが，Ruby の真の秘術が登場するのはまだまだこれからだ．

セルフスタディ 2 日目

2 日目の課題は少し難しい．Ruby を少しは使ってみて，本格的に取り組む準備は整ったはずだ．以下の課題では，少し分析的な思考が必要になる．

探してみよう：

- コードブロックを使った場合と使わない場合の両方について，ファイルにアクセスするコードを書く．コードブロックの利点は？
- ハッシュを配列に変換するにはどうすればよいか？ また，逆に配列をハッシュに変換する方法は？
- ハッシュの各要素について繰り返すにはどうすればよいか？

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- Ruby の配列はスタックとしても使える．スタック以外に，配列で実現可能なよくあるデータ構造体を挙げよ．

試してみよう：

- 最初に，each だけを用いて，16 個の数値と 4 個の数値の配列の中身を同時に出力せよ．次に，同じことを Enumerable の each_slice を用いて実行せよ．
- Tree クラスは面白いクラスだったが，きれいなインタフェースを用いて新しいツリーを指定することはできなかった．そこで，initializer にハッシュと配列が入れ子になった構造体を指定できるようにせよ．具体的には，次のようなツリーを指定できるようにしたい．`{'grandpa' => { 'dad' => {'child 1' => [], 'child 2' => [] }, 'uncle' => {'child 3' => [], 'child 4' => [] } } }`．
- ファイル内で，あるフレーズを含むすべての行を出力する簡単な grep を書け．簡単な正規表現でマッチングを行い，ファイルから行を読み出す必要がある（この処理は Ruby では驚くほど簡単に書ける）．必要なら行番号も出力してみるとよい．

2.4　3 日目：真剣な変更

メリー・ポピンズの醍醐味は，彼女が家事を楽しくこなし，情熱と想像力で家族一人ひとりの心を変えて，全体として良い家族を作り上げていくところにある．少しスピードをゆるめ，確実に理解できるように安全運転で，他の言語で既にやり方が分かっていることを Ruby で実行してみることもできるだろう．でも，言語の見た目や動作の仕方を変えてみると，プログラミングを楽しくする魔法をつかむことができる．本書では，章ごとに，簡単ではないけれど，その言語を使うとうまく解決できる問題を紹介する．この Ruby の章では，メタプログラミングを紹介することにする．

メタプログラミングとは，プログラムを書くプログラムを作成することを意味する．Rails の中核部分を成す ActiveRecord フレームワークでは，データベースのテーブルに接続するためのクラスを構築するのに便利な言語をメタプログラミングによって実装している．例えば Department の ActiveRecord クラスは次のようになる．

```
class Department < ActiveRecord::Base
  has_many :employees
  has_one :manager
end
```

has_many と has_one は，has_many 関係を確立するために必要なすべてのインスタンス変数とメソッドを追加する Ruby のメソッドだ．このクラスの仕様は，他のデータベースフレームワークによくあるような余計な情報や理解の妨げになる情報がすべて除外されているため，英語のように楽々と読める．メタプログラミングに利用できるツールをいくつか見てみることにしよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

オープンクラス

オープンクラスはここまでの例でも少し利用した．クラスの定義はいつでも変更できる．通常は振る舞い（メソッド）を追加するときにクラスを変更する．以下に，Railsフレームワークから引用した典型的な例を示す．この例ではNilClassにメソッドを追加している．

▶ code/ruby/blank.rb

```
class NilClass
  def blank?
    true
  end
end

class String
  def blank?
    self.size == 0
  end
end

["", "person", nil].each do |element|
  puts element unless element.blank?
end
```

先頭行のclassによってクラスが定義される．クラスがいったん定義されたら，以降の呼び出しはそのクラスに対する変更となる．上のコードでは，既存の2つのクラスNilClassとStringにblank?というメソッドを追加している．指定された文字列の状態を確認するときには，その文字列が空白かどうかを確認することが多い．大半の文字列は，値，空文字，nilのいずれかだ．このblank?というイディオムを使えば，空文字列に相当する2つのケースでtrueが返るため，同時に手早くチェックすることができる．Stringが指しているクラスは何でも構わない．そのクラスがblank?メソッドをサポートしていれば，うまく動作する．アヒルのように歩きアヒルのように鳴く動物がいたら，それはアヒルだ．その動物がアヒルかどうかを確認するためにいちいち血液検査をする必要はない．

何が起こっているのか見てみよう．あなたが切れ味鋭いメスを要求すると，Rubyは喜んでそのメスを提供する．この例では，オープンクラスでStringとNilの2つのクラスを再定義した．やろうと思えば，Class.newを再定義してRubyを完全に動作不能にしてしまうこともできる．それと引き替えに得られるのは「自由」だ．あらゆるクラスやオブジェクトをいつでも再定義できる自由があれば，驚くほど読みやすいコードを書くことができる．ただし，自由と権力には責任がついて回る．

オープンクラスは，ドメイン固有言語（自分の専門領域をコード化する言語）を構築するときに便利だ．自分のビジネス分野で使う言語で単位を表現するときによく使用される．例えば，すべての距離をインチで表現するAPIを考えてみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/ruby/units.rb

```
class Numeric
  def inches
    self
  end

  def feet
    self * 12.inches
  end

  def yards
    self * 3.feet
  end

  def miles
    self * 5280.feet
  end

  def back
    self * -1
  end

  def forward
    self
  end
end

puts 10.miles.back
puts 2.feet.forward
```

オープンクラスでは，最小限のシンタックスでこの種の変更をサポートできる．が，他の方法を使って，Ruby をさらに拡張することもできる．

method_missing を使う

Ruby は，目的のメソッドが見つからないと，特殊なデバッグ用メソッドを呼び出して診断情報を出力する．この振る舞いのおかげでデバッグが容易になる．が，それだけではない．ときには，この機能を利用して，思いもかけないような高度な振る舞いをさせることができる．必要なのは，method_missing を上書きすることだけだ．ローマ数字を表現する API を考えてみよう．これは，例えば Roman.number_for "ii"などの API を用いたメソッド呼び出しによって簡単に実現できる．実際，この方法でも悪くはない．途中に括弧やセミコロンを挿入する必要はない．しかし Ruby ではもっとうまい方法がある．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/ruby/roman.rb

```
class Roman
  def self.method_missing name, *args
    roman = name.to_s
    roman.gsub!("IV", "IIII")
    roman.gsub!("IX", "VIIII")
    roman.gsub!("XL", "XXXX")
    roman.gsub!("XC", "LXXXX")

    (roman.count("I") +
     roman.count("V") * 5 +
     roman.count("X") * 10 +
     roman.count("L") * 50 +
     roman.count("C") * 100)
  end
end

puts Roman.X
puts Roman.XC
puts Roman.XII
puts Roman.X
```

このコードは，method_missing の極めて美しい使用例だ．内容は明快かつ簡単．まず method_missing をオーバーライドする．その際，メソッドの名前と引数を入力引数として受け取る．ここではメソッド名だけを使う．最初に，メソッド名を String に変換する．次に，iv や ix などの特殊なケースを，カウントしやすい文字列で置換する．そしてローマ数字を数え，その数字に対応する値を掛ける．API は極めて簡単だ．Roman.i と書くだけでよい．いちいち Roman.number_for "i"と書く必要はない．

ここで，コストを考えてみよう．この Roman クラスでは，メソッドが見つからなくても Ruby は知らせてくれなくなるので，デバッグが大変難しくなる．厳密なエラーチェックを実行して，有効なローマ数字を受け取ったかどうかを確認する機能がどうしても欲しいところだ．何を探しているのか分からない状態では，Roman の ii メソッドの実装を見つけるのは難しい．とはいえ，これが Ruby の道具箱に入っている鋭いメスであることは間違いない．賢明に使うようにしたい．

モジュール

Ruby で最も人気のあるメタプログラミングスタイルはモジュールだ．def や attr_accessor などは，文字どおり数行のモジュールとして実装できる．クラス定義も驚くほど簡単に拡張できる．よく知られている手法を用いて，ドメイン固有言語（DSL）を設計し，自身のクラスを定義することもできる[*6]．DSL ではモジュール内にメソッドを定義し，クラスを管理するために必要なすべてのメソッドと定数を追加する．

ここではまず，一般的なスーパークラスを使ってコードサンプルを分解して説明していくことにする．以下に，メタプログラミングを使って構築するのに適したクラスの例

[*6] DSL のテクニックを使うと，特定のドメインで使う言語を作ることができる．Ruby で最もよく知られている例として，ActiveRecord 永続化フレームワークがある．このフレームワークでは，DSL を使ってクラスをデータベースのテーブルにマッピングしている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

を示す．クラス名に基づいて CSV ファイルを開く簡単なプログラムだ．

▶ code/ruby/acts_as_csv_class.rb

```
class ActsAsCsv
  def read
    file = File.new(self.class.to_s.downcase + '.txt')
    @headers = file.gets.chomp.split(', ')

    file.each do |row|
      @result << row.chomp.split(', ')
    end
  end

  def headers
    @headers
  end

  def csv_contents
    @result
  end

  def initialize
    @result = []
    read
  end
end

class RubyCsv < ActsAsCsv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

この基本的なクラスでは 4 つのメソッドを定義している．headers と csv_contents は，インスタンス変数の値を返す簡単なアクセサだ．initialize は，読み取り結果を初期化する．大半の処理は read で行う．read メソッドは，ファイルを開き，見出しを読み取り，改行文字を切り落として，個々のフィールドに分割する．これを行単位で繰り返して，各行の内容を配列に格納する．この CSV ファイルの実装は，引用符などの特殊なケースに対応していないので完全ではないが，大体のイメージは分かる．

次の手順では，ファイルを読み取り，その振る舞いをクラスに結びつける．それにはマクロと呼ばれるモジュールメソッドを使う．マクロは環境の変化に応じてクラスの動作を変更する．この例では，我々のマクロはクラスを開き，CSV ファイルに関するすべての動作をダンプしている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/ruby/acts_as_csv.rb

```
class ActsAsCsv
  def self.acts_as_csv

    define_method 'read' do
      file = File.new(self.class.to_s.downcase + '.txt')
      @headers = file.gets.chomp.split(', ')

      file.each do |row|
        @result << row.chomp.split(', ')
      end
    end

    define_method "headers" do
      @headers
    end

    define_method "csv_contents" do
      @result
    end

    define_method 'initialize' do
      @result = []
      read
    end
  end
end

class RubyCsv < ActsAsCsv
  acts_as_csv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

メタプログラミングが行われているのは acts_as_csv マクロ内である．そこでは，対象とするクラスに追加する必要があるすべてのメソッドについて define_method を呼び出している．これにより，対象とするクラスが acts_as_csv を呼んだときに，対象クラス上で 4 つのメソッドがすべて定義される．

したがって acts_as は，継承を利用しても簡単に追加できるいくつかのメソッドを追加しただけで，ほかには何もしていない．この設計はあまり改善とは思えないかもしれないが，ここからが面白いところだ．同じ動作をモジュールで実装するとどうなるか見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/ruby/acts_as_csv_module.rb

```
module ActsAsCsv

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods
    def acts_as_csv
      include InstanceMethods
    end
  end

  module InstanceMethods

    def read
      @csv_contents = []
      filename = self.class.to_s.downcase + '.txt'
      file = File.new(filename)
      @headers = file.gets.chomp.split(', ')

      file.each do |row|
        @csv_contents << row.chomp.split(', ')
      end
    end

    attr_accessor :headers, :csv_contents

    def initialize
      read
    end

  end

end

class RubyCsv  # no inheritance! You can mix it in
  include ActsAsCsv
  acts_as_csv
end

m = RubyCsv.new
puts m.headers.inspect
puts m.csv_contents.inspect
```

Ruby は，モジュールが他のモジュールに include されるたびに，included メソッドを呼び出す．ここで，クラスはモジュールであることに注意してほしい．この例の included メソッドでは，base と呼ばれる対象クラス（RubyCsv クラスのこと）を拡張して，そのモジュールで RubyCsv にクラスメソッドを追加している．追加している唯一のクラスメソッドは acts_as_csv だ．そしてこのメソッドが，クラスを開き，すべてのインスタンスメソッドを include する．つまり，プログラムを書くプログラムを作成していることになる．

こうしたメタプログラミングで面白いのは，アプリケーションの状態に応じてプログラ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:34:37 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ムが変化する点だ．ActiveRecord では，メタプログラミングを使用して，データベースの列と同じ名前のアクセサを動的に追加している．builder など，一部の XML フレームワークでは，ユーザが method_missing を使用してカスタムのタグを定義し，美しいシンタックスを実現できる．シンタックスが美しいほど，コードの読み手は，シンタックスを気にせず，コードの真意に近づくことができる．それこそが Ruby の強みである．

3 日目で学んだこと

3 日目は，Ruby を使って独自のシンタックスを定義し，クラスを動的に変更する方法を学んだ．こうしたプログラミング手法をメタプログラミングと呼ぶ．あなたが書くコードのすべての行には，2 種類の読み手が存在する．つまり，コンピュータと人間だ．インタープリタやコンパイラによるエラーチェックに合格することと，分かりやすいコードを書くこと，この両者のバランスをとるのはときとして難しい．メタプログラミングを使用すれば，有効な Ruby のシンタックスと（実際にコードで使用する）文との間のギャップを埋めることができる．

Ruby の最高のフレームワークのなかには，Builder や ActiveRecord など，読みやすさを実現するために，メタプログラミングの手法を多用しているものがある．オープンクラスを使って，String オブジェクトと nil で blank?メソッドが使えるようにダックタイピングによるインタフェースを構築した．これにより，よくあるシナリオでのコードの煩雑さを大幅に削減できる．同様の手法を使ったコードも紹介した．method_missing を使って，ローマ数字を出力する美しいコードを書いた．最後に，CSV ファイルを指定するドメイン固有言語を，モジュールを使って定義した．

セルフスタディ 3 日目

試してみよう：

each メソッドが CsvRow オブジェクトを返すように，CSV アプリケーションを変更せよ．その CsvRow の method_missing を使って，与えられた見出しの列の値を返すようにせよ．

例えば，次のようなファイルがあるとする．

```
one, two
lions, tigers
```

このとき，次のような API が使えるようにせよ．

```
csv = RubyCsv.new
csv.each {|row| puts row.one}
```

この出力結果は"lions"となる．

2.5 Ruby のまとめ

この章では多くの基本事項について学んだ．メリー・ポピンズとの類似点に気づいていただけたことと思う．筆者は Ruby カンファレンスで何度も講演したことがあるが，講

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

演後，多くの Ruby 愛好者から Ruby でプログラミングするのは面白いという感想を聞かされた．C++，C#，Java，その他の C 系の言語を開発に使ってきた業界にとって，Ruby はさわやかな新風なのだ．

長所

Ruby は純粋なオブジェクト指向性を備えているため，一意で一貫性のある方法でオブジェクトを扱うことができる．ダックタイピングは，オブジェクトの継承階層ではなく，オブジェクトが何をサポートしているかに基づく，より真にポリモーフィズム的な設計を可能とする．Ruby のモジュールとオープンクラスを使用すると，プログラマは，クラスでの通常のメソッドやインスタンス変数の定義を超えたシンタックスに動作を結び付けることができる．

Ruby はスクリプト言語として理想的である．また，スケーラビリティ要件がそれほど厳しくなければ，Web 開発言語としても理想的である．生産性は極めて高い．ただし，高い生産性を可能にしているいくつかの機能のせいで，Ruby はコンパイルが難しく，パフォーマンスが低い．

スクリプティング

Ruby は素晴らしいスクリプト言語である．2 つのアプリケーションを結びつけるのり付け用のコード，Web ページを検索して株価や書籍の値段を探すスパイダー，ローカルのビルド環境や自動テストの実行などは，Ruby の優れた機能を発揮できる使用例だ．

Ruby は，大半の主要なオペレーティングシステムでサポートされている言語なので，スクリプト環境には格好の選択肢となる．Ruby の標準配布物には広範なライブラリが含まれているうえ，膨大な数の gem（あらかじめパッケージされたプラグイン）も入手可能となっており，CSV ファイルの読み込み，XML ファイルの処理，低レベルのインターネット API の操作などを実行できる．

Web 開発

Rails は，現時点で既に，これまでで最も成功を収めた Web 開発フレームワークの 1 つとなっている．Rails の設計は，よく知られているモデル・ビュー・コントローラ（MVC）のパラダイムに基づいている．データベースの命名やアプリケーションの命名に関して多数の規則を設けることにより，典型的なアプリケーションであれば，数行のコンフィギュレーションを書くだけでビルドできてしまう．また，本番環境で発生する難しい問題に対応するためのプラグインも用意されている．以下に Rails の利点をまとめておく．

- Rails アプリケーションの構造は常に一貫性があり，分かりやすい．
- 移行時には，データベーススキーマの変更にも対応している．
- 明確に文書化された規則により，コンフィギュレーション用のコーディング量が少なくて済む．
- さまざまなプラグインを利用できる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

市場までのリードタイム

Ruby と Rails の生産性は，Rails が成功を収めた重要な要因であると筆者は考えている．2000 年代半ばには，サンフランシスコで石を投げれば，Rails を採用している新興企業の社員に必ず当たるというぐらい，Rails は普及していた．現在でも，筆者の会社を含め，こうした企業では Ruby は大きな成果を収めている．美しいシンタックスと，プログラマ，ツール関連，プラグイン関連の各種コミュニティの組み合わせは，極めて強力だ．Web 閲覧者の郵便番号を検索する gem，半径 50 マイルの範囲内のすべての住所コードを計算する gem などがすぐに見つかる．画像やクレジットカードの処理，Web サービスの利用，多数のプログラミング言語とのやり取りも可能だ．

多くの大規模な商用 Web サイトで，Ruby と Ruby on Rails が採用されている．Twitter のオリジナル版の実装にも Ruby が採用され，Ruby の並外れた生産性のおかげで同サイトは巨大サイトへと成長を遂げることができた．最終的には，Twitter のコア部分は Scala で書き直された．この点に関して 2 つの教訓がある．1 つは，Ruby は，存続可能な製品を短期間で市場に送り出すという意味においては最適な言語であるという点．もう 1 つは，Ruby のスケーラビリティはいくつかの点で制限されているという点だ．

分散トランザクション，フェールセーフのメッセージング，国際化を行っている格式張った大企業では，Ruby の役割は少し限定されると思われがちだが，こうした処理はすべて Ruby でも実行できる．適切なアプリケーションフレームワークとスケーラビリティに関わる懸念は根拠のないものではないこともあるが，あまりに多くの人々が，期限内にソフトウェアを完成させる目処さえ立たないのに，eBay 並みのシステムを構築できるほどに高いスケーラビリティが必要だと思い込んでいる．多くの企業が直面している市場までのリードタイム要件に応えるには，Ruby でも十分すぎることが多い．

短所

すべてのアプリケーション向けに完璧な機能を備えた言語など存在しない．Ruby にも制限はある．以下では Ruby の主な制限について概説する．

パフォーマンス

Ruby 最大の弱点といえば，パフォーマンスである．もちろん高速化は図られている．バージョン 1.9 では，一部のケースで処理速度が 10 倍にもなっている．Evan Phoenix 氏作の新しい Ruby の仮想マシン Rubinius では，ジャストインタイム・コンパイラを使って Ruby をコンパイルできる．この方法では，インタープリタによる特定のコードブロックの使用パターンを見て，どのコードが再度必要となる可能性が高いかを予測する．この方法は，Ruby のように，シンタックスだけでは，コンパイルに必要な情報が十分に得られない言語ではうまく動作する．クラス定義はいつでも変更される可能性があるということを思い出してほしい．

それでも Matz の考え方は極めて明解だ．彼はプログラマがコードを書きやすくなるよう頑張っている．言語のパフォーマンスを最適化するのは二の次なのだ．オープンクラス，ダックタイピング，`method_missing` など，Ruby が備えている機能の多くが，コンパイルとそれに伴うパフォーマンスの向上を実現するツールの効力を無効にしてしまう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

並行性と OOP

オブジェクト指向プログラミングには重大な制限がある．モデルの前提が，状態をカプセル化して隠ぺいし，振る舞いをメソッドとして公開するという考え方に依存しているが，状態は変わる可能性がある．このプログラミング方式では，並行性に関して重大な問題を引き起こす．まず，重大なリソース競合が言語に組み込まれてしまう．最悪の場合，オブジェクト指向システムは，デバッグが不可能となり，並行環境で信頼性の高いテストを実行できなくなる．この章の執筆時点では，Rails チームは，並行性管理の問題に対する効果的な対応策をようやく検討し始めたところだ．

型の安全性

筆者はダックタイピングの固い信奉者だ．この型付け方式では一般に，抽象化が明確になり，コードもコンパクトで読みやすくなる．ただし，ダックタイピングには代償が伴う．静的型付けでは，シンタックスツリーの作成を容易にするさまざまなツールが利用できるため，統合化開発環境を提供できる．Ruby の IDE を構築するのは難しい．今までのところ，大半の Ruby 開発者は IDE を使っていない．筆者は IDE スタイルのデバッガが使えないと何度も嘆いたことがあるが，それは筆者だけではないことも分かっている．

終わりに

Ruby の強みは，そのシンタックスと柔軟性だ．主な弱点はパフォーマンスだが，大半の用途ではそこそこのパフォーマンスを達成している．全体として Ruby は，オブジェクト指向開発には卓越した言語である．しかるべきアプリケーションに使えば，Ruby は素晴らしい威力を発揮する．また他のツールと同様，しかるべき問題の解決に使えば，失望させられることはまずないだろう．よく注意して，Ruby のちょっとしたマジックを探してみてほしい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第3章

Io

問題は「何をするか」ではない．「何をしないか」だ．

フェリス・ビューラー

Io を紹介しよう．Ruby 同様，Io は掟破りの言語だ．若くて，ずるがしこくて，分かりやすいが，予測不可能．フェリス・ビューラー[*1][*2]を思い浮かべてもらうとよい．フェリスと楽しく過ごしたければ，彼に街を案内させるのがいい．Io も同じだ．フェリスは何でも一度はやってみるタイプ．あなたに人生で最高の体験をさせてくれるかもしれないし，親父さんの車を壊してしまうかもしれないし，その両方かもしれない．いずれにしろ退屈はしないだろう．冒頭に引用したフェリスの言葉どおり，あなたを邪魔するルールはそれほど多くはない．

3.1 Io とは

Io は 2002 年に Steve Dekorte 氏によって開発された．I は常に大文字で書き，その後に小文字の o が続く．Io は Lua や JavaScript と同様にプロトタイプ言語だ．つまり，すべてのオブジェクトは別のオブジェクトのクローンである．

Io は元々，Steve がインタープリタの動作方法を理解するための練習として書いた言語であり，趣味レベルの言語として生まれた．現在でもそのサイズは極めてコンパクトだ．Io のシンタックスはものの 15 分で，基本的な言語の仕組みは 30 分もあれば覚えられる．驚くようなことは何もない．しかしライブラリを覚えるには少し時間がかかるだろう．Io の複雑さと豊富な機能はライブラリによって実現されている．

現在，Io のコミュニティは，小さな仮想マシンと高い並行性を備えた組み込み可能な言語としての Io に焦点を合わせている．Io の強みは，さまざまにカスタマイズ可能なシンタックスと関数，および強力な並行性モデルだ．シンタックスのシンプルさとプロトタイプ・プログラミングモデルに注目してほしい．筆者は Io について学んだあと，JavaScript の動作方法についての理解がぐっと深まった．

*1 『フェリスはある朝突然に』（*Ferris Bueller's Day Off*）．DVD．John Hughes 監督作品．1986 年．カリフォルニア州ハリウッド．Paramount，1999 年．

*2 ［訳注］高校生のフェリスが学校を休んで，友達を誘って街へ遊びに出かけた一日の顛末を描いた映画．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

3.2　1 日目：学校をさぼって，気ままに過ごす

Io との最初の出会いは，他の言語との出会いと変わらない．ちゃんと慣れるには，少しキーボードを叩いてみる必要がある．真面目な歴史の授業の前に，廊下で気楽な雑談をしておけば，理解がずっと楽になるというものだ．じゃあ堅苦しい説明は抜きにして，面白いものから始めてみよう．

名前が必ずしも実体を表すとは限らないが，Io という名前からは多くを知ることができる．Io などという単純な言葉を言語の名前にするなど向こう見ずだが（*Io* を Google で検索したことがあるだろうか）[*3]，いかした名前でもある．Io はわずか 2 文字であり，両方とも母音である．Io も，その名前と同様，向こう見ずな側面と優れた側面の両方を備えている．言語のシンタックスは，Io という名前と同様，単純でローレベルに設計されている．Io のシンタックスは，単純にメッセージをチェーン接続したもので，各メッセージはオブジェクトを返す．各メッセージには括弧付きで任意指定の引数を指定できる．Io では，すべてのものは別のレシーバを返すメッセージである．キーワードはないが，キーワードのように振る舞う文字がいくつかある．

Io では，クラスとオブジェクトの両方を意識する必要がない．もっぱらオブジェクトだけを扱い，必要に応じてオブジェクトを複製（クローニング）する．これらのクローンはプロトタイプと呼ばれる．Io は，本書で取り上げる唯一のプロトタイプベースの言語である．プロトタイプベースの言語では，すべてのオブジェクトが既存のオブジェクトのクローンとなる．クラスからオブジェクトが生成されることはない．Io を学んで得られるものは，オブジェクト指向の Lisp を学んだときに得られるものにとても近い．Io の将来を判断するには時期尚早だが，シンタックスが簡単なので，Io にも十分に他の言語と競争できる力はある．3 日目で取り上げる並行性ライブラリはよく練られており，メッセージシンタックスはエレガントで強力だ．リフレクションはいたるところで使われている．

手始めに

さっそくインタープリタを開いて，気軽に遊んでみよう．Io は `http://iolanguage.com` に置いてある．ダウンロードしてインストールしてみよう．インタープリタを開くには，`io` と入力する．おなじみの “Hello, World” プログラムを書いてみる．

```
Io> "Hi ho, Io" print
Hi ho, Io==> Hi ho, Io
```

何が起こっているのかお分かりだと思う．`print` メッセージを文字列`"Hi ho, Io"`に送信している．レシーバが左側，メッセージが右側だ．シンタックスシュガーはほとんど見あたらない．単に，オブジェクトにメッセージを送信しているだけだ．

Ruby では，クラスに対して `new` を呼ぶことで新しいオブジェクトを作成する．新しい種類のオブジェクトを作成するには，クラスを定義する．Io では，クラスとオブジェクトを区別しない．既存のオブジェクトを複製することで新しいオブジェクトを作成する．既存のオブジェクトのことをプロトタイプという．

[*3] 「Io 言語」と入力することをお勧めする．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
batate$ io
Io 20090105
Io> Vehicle := Object clone
==>  Vehicle_0x1003b61f8:
  type             = "Vehicle"
```

Objectはルートレベルのオブジェクトだ．これにcloneメッセージを送信すると，新しいオブジェクトが返される．ここでは，生成されたオブジェクトをVehicleに代入している．ここで，Vehicleはクラスではない点に注意してほしい．Vehicleはオブジェクトを作成するために使うひな形（テンプレート）ではなく，Objectプロトタイプを基に作成されたオブジェクトである．ではVehicleと対話してみよう．

```
Io> Vehicle description := "Something to take you places"
==> Something to take you places
```

オブジェクトはスロットと呼ばれるキーと値の組を複数持つことができる．スロットのコレクションはハッシュと考えてよい．各スロットはキーで参照する．スロットへの代入には:=を使う．スロットが存在しない場合は，作成される．スロットへの代入に=を使うこともできる．こちらを使った場合は，スロットが存在しないと例外が発生する．上の例ではdescriptionという名前のスロットを作成している．

```
Io> Vehicle description = "Something to take you far away"
==> Something to take you far away
Io> Vehicle nonexistingSlot = "This won't work."

  Exception: Slot nonexistingSlot not found.
    Must define slot using := operator before updating.
  ---------
  message 'updateSlot' in 'Command Line' on line 1
```

スロットの値を取得するには，スロットの名前をオブジェクトに送信する．

```
Io> Vehicle description
==> Something to take you far away
```

実際には，オブジェクトは単なるスロットの集まりと考えて構わない．Vehicleのすべてのスロットの名前を確認するには，次のようにする．

```
Io> Vehicle slotNames
==> list("type", "description")
```

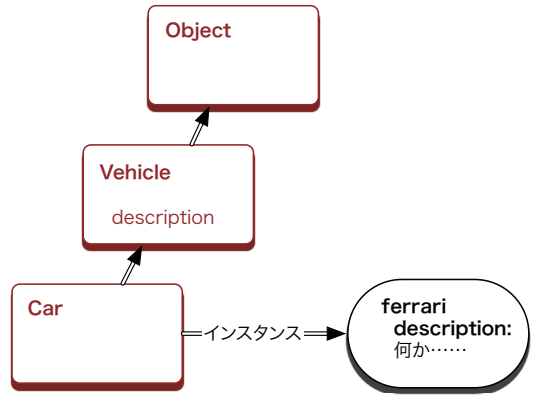
slotNamesメソッドをVehicleに送信して，スロット名のリストを取得している．Vehicleには2つのスロットがある．descriptionスロットについては既に見たが，もう1つ，typeスロットがある．すべてのオブジェクトはtypeに答えることができる．

```
Io> Vehicle type
==> Vehicle
Io> Object type
==> Object
```

type（タイプ）については，もう少し後で触れる．とりあえずここでは，typeは扱っているオブジェクトの種類を表すことだけ知っておけばよい．タイプはオブジェクトで

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社



▲ 図 3.1　オブジェクト指向設計

あってクラスではないことも忘れないように．以下に，ここまで述べたことをまとめておく．

- オブジェクトは他のオブジェクトを複製（クローニング）することによって作成する．
- オブジェクトはスロットの集まりである．
- スロットの値を取得するにはメッセージを送信する．

Io は簡単で面白いということはお分かりいただけたことと思う．でも，まだまだほんの入口部分にすぎない．次に，継承について説明しよう．

オブジェクト，プロトタイプ，継承

この節では継承について説明する．車（Vehicle に属する）が与えられたとき，車のインスタンスである ferrari オブジェクトをモデル化する方法を考えてみよう．オブジェクト指向言語では，図 3.1「オブジェクト指向設計」に示すような操作を実行する．

同じ問題をプロトタイプ言語で解決する方法を考えてみる．いくつかオブジェクトを追加作成する必要がある．まずオブジェクトを 1 つ作成しよう．

```
Io> Car := Vehicle clone
==>  Car_0x100473938:
  type             = "Car"

Io> Car slotNames
==> list("type")
Io> Car type
==> Car
```

Io 風に説明すると「clone メッセージを Vehicle プロトタイプに送信して新しいオブ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ジェクト Car を作成した」となる．Car に description を送信してみよう．

```
Io> Car description
==> Something to take you far away
```

Car には description スロットは存在しない．Io は description メッセージをプロトタイプに転送し，Vehicle に description スロットがあるのを見つける．この動作は極めてシンプルだが，非常に強力だ．ここでもう 1 台車を作成してみる．今度は作成した車を ferrari に代入する．

```
Io> ferrari := Car clone
==>  Car_0x1004f43d0:

Io> ferrari slotNames
==> list()
```

なるほど．type スロットも存在しない．慣習上，Io ではタイプの先頭文字に大文字を使う．ここで type スロットを呼び出すと，プロトタイプのタイプが返される．

```
Io> ferrari type
==> Car
```

以上が Io のオブジェクトモデルの動作原理だ．オブジェクトはスロットの入れ物にすぎない．スロットを取得するには，スロット名をオブジェクトに送信する．スロットが存在しない場合は，親が呼び出される．理解する必要があるのはこれだけだ．クラスもメタクラスも存在しない．インタフェースやモジュールもない．ただオブジェクトが存在するだけだ（図 3.2「Io における継承」(次ページ)を参照）．

Io のタイプは単なる便宜的な道具にすぎない．慣用語法的には，大文字で始まる名前を持つオブジェクトがタイプとなるため，Io は type スロットを設定する．同じ type を持つオブジェクトのクローンで小文字で始まるものは，単純に親の type スロットを呼び出す．タイプは，Io プログラマがコードを整理しやすくするための単なるツールにすぎない．

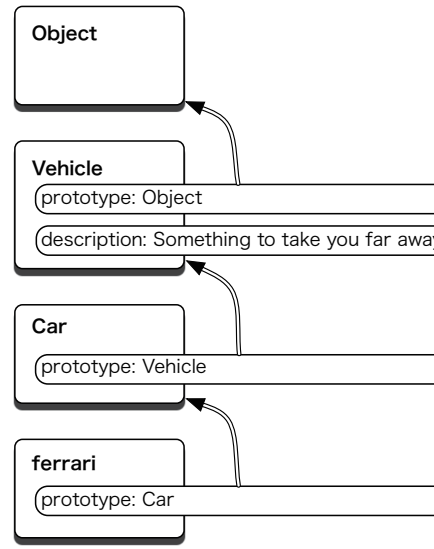
ferrari をタイプにしたければ，次のように大文字で始めればよい．

```
Io> Ferrari := Car clone
==> Ferrari_0x9d085c8:
type = "Ferrari"

Io> Ferrari type
==> Ferrari

Io> Ferrari slotNames
==> list("type")
Io> ferrari slotNames
==> list()
Io>
```

ferrari には type スロットがないが，Ferrari にはある点に注意してほしい．ここでは，タイプとインスタンスの違いを説明するために，言語機能をフルに使わず，簡単なコーディング規則を使用することにする．したがって，タイプとインスタンスは，先頭文

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 3.2　Io における継承

字が大文字か小文字かの違いだけで，それ以外の点はまったく同じように振る舞う．

Ruby や Java では，クラスはオブジェクトを作成するためのひな形だ．`bruce = Person.new` と書くと，Person クラスから新しい person オブジェクトが作成される．両者は，クラスとオブジェクトというまったく異なるエンティティだ．Io では様子が異なる．Io では，`bruce := Person clone` と書くと，Person というプロトタイプから bruce というクローンが作成される．bruce も Person もオブジェクトだ．Person はタイプだ．なぜなら Person には type スロットがあるからだ．それ以外の点では Person と bruce は同じである．では振る舞いの説明に移ろう．

メソッド

Io では簡単にメソッドを作成できる．次に例を示す．

```
Io> method("So, you've come for an argument." println)
==> method(
    "So, you've come for an argument." println
)
```

メソッドは，他の任意のタイプのオブジェクトと同様，オブジェクトである．メソッドのタイプを取得するには次のように書く．

```
Io> method() type
==> Block
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:35:49 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

メソッドはオブジェクトなので，スロットに代入することができる．

```
Io> Car drive := method("Vroom" println)
==> method(
    "Vroom" println
)
```

スロットにメソッドが代入されている場合，そのスロットを呼び出すとメソッドが呼び出される．

```
Io> ferrari drive
Vroom
==> Vroom
```

信じられないかもしれないが，Io の中心となる構成原理についての説明はこれだけだ．基本的なシンタックスについては触れたし，タイプとオブジェクトも定義できるはずだ．データと振る舞いをオブジェクトに追加するには，その内容をオブジェクトのスロットに代入すればよい．これ以外のことを行うには，ライブラリについて学習する必要がある．

少し掘り下げて考えてみよう．スロットの内容を取得するには次のように書く．これはスロットの内容が変数でもメソッドでも同じだ．

```
Io> ferrari getSlot("drive")
==> method(
    "Vroom" println
)
```

getSlot は，該当するスロットが存在しないと親のスロットを返す．

```
Io> ferrari getSlot("type")
==> Car
```

特定のオブジェクトのプロトタイプを取得するには次のように書く．

```
Io> ferrari proto
==>  Car_0x100473938:
  drive             = method(...)
  type              = "Car"

Io> Car proto
==>  Vehicle_0x1003b61f8:
  description       = "Something to take you far away"
  type              = "Vehicle"
```

これらは，ferrari と Car をクローンするときに使ったプロトタイプだ．便宜上，カスタムのスロットも表示される．

Lobby というマスター名前空間には，すべての名前付きオブジェクトが含まれる．ここまでコンソールで実行したすべての代入式，およびその他数個の代入式が Lobby に登録されている．Lobby の内容は次のようにして確認できる．

```
Io> Lobby
==>  Object_0x1002184e0:
  Car               = Car_0x100473938
  Lobby             = Object_0x1002184e0
  Protos            = Object_0x1002184e0
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Vehicle          = Vehicle_0x1003b61f8
exit             = method(...)
ferrari          = Car_0x1004f43d0
forward          = method(...)
```

我々が定義したオブジェクトのほかに，exit の実装，forward，Protos が表示されているのが分かる．

プロトタイプ・プログラミングのパラダイムは非常に明解だ．以下に基本的なルールを示す．

- すべてのモノはオブジェクトである．
- オブジェクトとのすべてのやり取りはメッセージを介して行う．
- クラスをインスタンス化するのではなく，他のオブジェクト（プロトタイプという）を複製する（クローンを作成する）．
- オブジェクトは自身のプロトタイプを記憶している．
- オブジェクトにはスロットがある．
- スロットにはオブジェクト（メソッドオブジェクトを含む）が格納される．
- メッセージはスロットが保持している値を返したり，スロットに格納されているメソッドを呼び出したりする．
- オブジェクトは，自分が応答できないメッセージを自分のプロトタイプに送信する．

これでほぼすべてだ．スロットやオブジェクトはいつでも確認および変更できるので，かなり高度なメタプログラミングも可能だ．その前にまず，Io の構成ブロックの次の階層，すなわちコレクションについて見ておく必要がある．

リストとマップ

Io には，コレクションの種類がいくつかある．リストは任意のタイプのオブジェクトの順序付きコレクションだ．List はすべてのリストのプロトタイプであり，Map は，Ruby のハッシュのようなキーと値のペアのプロトタイプだ．リストは次のようにして作成する．

```
Io> toDos := list("find my car", "find Continuum Transfunctioner")
==> list("find my car", "find Continuum Transfunctioner")

Io> toDos size
==> 2

Io> toDos append("Find a present")
==> list("find my car", "find Continuum Transfunctioner", "Find a present")
```

リストには簡単な表現方法がある．Object でサポートされている list メソッドを使う方法だ．list メソッドは，渡された引数をすべて使って 1 つのリストを作る．list メソッドを使うと，次のように簡単にリストを作成できる．

```
Io> list(1, 2, 3, 4)
==> list(1, 2, 3, 4)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

List には数学用の便利なメソッドや，リストを他のデータ型（スタックなど）として使うためのメソッドも用意されている．

```
Io> list(1, 2, 3, 4) average
==> 2.5

Io> list(1, 2, 3, 4) sum
==> 10

Io> list(1, 2, 3) at(1)
==> 2

Io> list(1, 2, 3) append(4)
==> list(1, 2, 3, 4)

Io> list(1, 2, 3) pop
==> 3

Io> list(1, 2, 3) prepend(0)
==> list(0, 1, 2, 3)

Io> list() isEmpty
==> true
```

Io でよく使われるもう 1 つのコレクションとしてマップがある．Io のマップは Ruby のハッシュに似ている．シンタックスシュガーはないので，次のような API を用いて操作する．

```
Io> elvis := Map clone
==>  Map_0x115f580:

Io> elvis atPut("home", "Graceland")
==>  Map_0x115f580:

Io> elvis at("home")
==> Graceland

Io> elvis atPut("style", "rock and roll")
==>  Map_0x115f580:

Io> elvis asObject
==>  Object_0x11c1d90:
  home              = "Graceland"
  style             = "rock and roll"

Io> elvis asList
==> list(list("style", "rock and roll"), list("home", "Graceland"))

Io> elvis keys
==> list("style", "home")

Io> elvis size
==> 2
```

ハッシュというのは実は Io のオブジェクトと構造がよく似ている．Io のオブジェクトでも，キーとしてのスロットに値が関連付けられているからだ．スロットの組み合わせ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

を簡単にオブジェクトに変換できるというのは興味深い．

ここまで基本的なコレクションについて見てきた．実際に使ってみたくなったことと思う．その前に制御構造について見ておく必要がある．制御構造はブール値に依存する．

true, false, nil, singleton

Io の条件は，他のオブジェクト指向言語とほぼ同じだ．以下にいくつか例を示す．

```
Io> 4 < 5
==> true
Io> 4 <= 3
==> false
Io> true and false
==> false
Io> true and true
==> true
Io> true or true
==> true
Io> true or false
==> true
Io> 4 < 5 and 6 > 7
==> false
Io> true and 6
==> true
Io> true and 0
==> true
```

とても簡単だ．ここで 1 つ注意してほしい．0 は Ruby と同じで真だ．C のように偽にはならない．では true とは何だろうか．少し調べてみよう．

```
Io> true proto
==>  Object_0x200490:
                = Object_()
 !=             = Object_!=()
...

Io> true clone
==> true
Io> false clone
==> false
Io> nil clone
==> nil
```

これは面白い．true，false，nil はシングルトンなのだ．それらのクローンを作成すると，シングルトン値が返される．これと同じことは極めて簡単にできる．次のようにして自分でシングルトンを作成すればよい．

```
Io> Highlander := Object clone
==>  Highlander_0x378920:
  type            = "Highlander"

Io> Highlander clone := Highlander
==>  Highlander_0x378920:
  clone           = Highlander_0x378920
  type            = "Highlander"
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ここでは，clone メソッドを再定義して，Highlander[*4]を返すようにしている．通常なら，Io によってツリーの上層にリクエストが転送され，最終的には Object に行き着くところだ．これで，Highlander を使うと，次のような動作をするようになる．

```
Io> Highlander clone
==>  Highlander_0x378920:
  clone            = Highlander_0x378920
  type             = "Highlander"
Io> fred := Highlander clone
==>  Highlander_0x378920:
  clone            = Highlander_0x378920
  type             = "Highlander"

Io> mike := Highlander clone
==>  Highlander_0x378920:
  clone            = Highlander_0x378920
  type             = "Highlander"

Io> fred == mike
==> true
```

このように 2 つのクローンが等しくなる．次に示すように，これは一般には真ではない．

```
Io> one := Object clone
==>  Object_0x356d00:

Io> two := Object clone
==>  Object_0x31eb60:

Io> one == two
==> false
```

これで，Highlander は一人しか存在できない．Io は，足元をすくわれることもあるくらいに自由度が高い．この解法は，多少予想外ではあるが，シンプルかつエレガントだ．大急ぎでたくさんの情報を紹介したが，シングルトンを生成するようにオブジェクトの clone メソッドを変更するなど，かなり過激なこともできるくらいの知識を身に付けたことになる．

ただし注意してほしい．好き嫌いはあるにせよ，Io が面白いことは否定できない．Ruby と同様，Io にも好ましい部分とそうでない部分がある．例えば，任意のオブジェクトのスロットを自由に変更できる．それが言語を定義するオブジェクトであってもだ．次のような操作は行うべきではない．

```
Object clone := "hosed"
```

オブジェクトの clone メソッドをオーバーライドしているため，以降はオブジェクトを一切作成できなくなる．これは修正できない．元に戻すには，プロセスを終了する必要がある．同時に，この操作によって，短期間で驚くような振る舞いをさせることができ

[*4] ［訳注］Highlander（1986）：最後に残った一人が究極の力を得るといわれる戦士たちの戦いを描くファンタジー映画．邦題『ハイランダー 悪魔の戦士』．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

るようにもなる．オブジェクトを形作っている演算子とスロットに自由にアクセスできるため，数行の短いコードで見事にドメイン固有言語を作成できる．初日の締めくくりに，Io を開発した Steve Dekorte 氏とのインタビューを読んでほしい．

Steve Dekorte 氏へのインタビュー

Steve Dekorte 氏は，サンフランシスコを中心に活躍するフリーのコンサルタントだ．同氏は 2002 年に Io を考案した．Io の開発に関するさまざまな話を聞いた．

Bruce Tate　なぜ Io を書いたのですか？

Steve Dekorte　2002 年に私の友人である Dru Nelson が（Self にインスピレーションを得て）Cel という言語を開発したのですが，その際，言語の実装に関するフィードバックを私に依頼してきたのです．当時の私は，何か役に立つ意見を言えるほどにはプログラミング言語の動作について理解していないと感じていたので，理解を深めるために自分で小さな言語を書き始めたのです．それが Io に成長したというわけです．

Bruce Tate　Io のどの部分がご自身で最も気に入っていますか．

Steve Dekorte　簡単で一貫性のあるシンタックスとセマンティクスですね．このおかげで何が起こっているのかが容易に分かります．基本的な操作はすぐに覚えられます．私は記憶力が悪く，C のシンタックスや奇妙なセマンティクスをしょっちゅう忘れてしまって，そのたびに調べなおしています（編注：Steve は Io を C で実装した）．ですから，Io を使うときに，そういう面倒なことはやらずに済むようにしたいと考えました．
例えば，`people select(age > 20) map(address) println` というコードを見れば，何をしているのかすぐに分かります．人を年齢で絞り込み，住所を取得して，出力しています．
セマンティクスを簡素化すると，さらに柔軟性が増します．言語実装者本人も予想していなかったことを，プログラマの自由な発想で組み立てることができるようになります．例えば，1 つの答えを想定したパズルを解かせるビデオゲームもあれば，結末の決まっていないゲームもあります．結末のないゲームは，ゲームの設計者が想像もしなかったようなことをプレーヤーができてしまうので面白い．Io にもそれと似たところがあります．言語によっては，シンタックスショートカットを用意していることがあります．そのため，余分な構文解析ルールが必要になります．プログラミング言語でプログラムを書くとき，プログラマは頭の中でパーサー（構文解析器）を働かせる必要があります．言語が複雑であればあるほど，頭の中でより多くの構文解析を行う必要があります．パーサーの作業量が増えると，プログラマの負担が増えてしまいます．

Bruce Tate　Io の制限について教えてください．

Steve Dekorte　Io の柔軟性に伴うコストとして，多くのごく一般的な使い方でも処理速度が遅くなる点が挙げられます．その一方で，利点もいくつかあります．例えばコルーチン，非同期ソケット，SIMD のサポートなどです．これらの利点によって，従来のソケット別スレッド並列性や非 SIMD ベクタ演算などを用いて書かれた C のアプリケーション

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

より高速になることさえあります．
構文規則が単純すぎると，コードの見た目が単調で読みづらくなるという意見を何度かもらったことがあります．私も Lisp でよく似た経験をしたので，その点は理解できました．構文要素を増やせば手早く読むことができますから．Io を使い始めたユーザからは，構文が単純すぎるという声を聞くことがありますが，たいてい徐々に受け入れられるようになります．

Bruce Tate　Io が実際に使われている環境で，最も意外だったのはどんな環境ですか．

Steve Dekorte　数年の間に，衛星，ルータ設定言語，ビデオゲーム用のスクリプト言語などに Io が使われているという噂を聞いています．Pixar でも使っており，同社のブログで Io が取り上げられています．

初日はかなりハードスケジュールだった．この辺りで少し休憩しよう．ここまで学習したことを実際に練習してみてほしい．

1 日目で学んだこと

1 日目を終えたところだが，Io のかなりの部分を学んだ．ここまでで，Io の基本的な性質についてかなりの知識を得たことになる．プロトタイプ言語である Io は非常にシンプルなシンタックスを採用しており，それを使って Io 自体の新しい基本要素を構築できる．中核部分の要素でも，簡単なシンタックスシュガーさえ使われていない．ある意味，この複雑さを最小限に抑えたミニマルアプローチのせいで，コードを読むのに少し苦労することになる．

一方，シンタックスを最小限に抑えることによって利点も生まれる．シンタックス面で複雑なことは何もないので，特殊なルールや例外を覚える必要もない．1 つの文の読み方を理解してしまえば，すべての文を読むことができる．おかげで学習時間の大半をボキャブラリの習得に割くことができる．

このため，新しく Io を学び始めたプログラマの負担は大幅に軽減される．初心者が理解すべき点は次のとおりだ．

- いくつかの基本的なシンタックスルール
- メッセージ
- プロトタイプ
- ライブラリ

セルフスタディ 1 日目

Io の背景を調べるとき，答えを検索するのは少し骨が折れる．というのは，Io という言葉には膨大な数の異なる意味があるからだ．Google で検索するときは「Io 言語」と入力することをお勧めする．

探してみよう：

- Io のいくつかの問題点の例

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- 質問に答えてくれるIoコミュニティ
- Ioのイディオムに関するスタイルガイド

確認してみよう:

- `1 + 1`を評価してから，`1 + "one"`を評価する．Ioは強く型付けされた言語か，それとも弱く型付けされた言語か？ コードを書いて答えを確かめてみてほしい．
- 0は真か偽か？ 空文字列はどうか？ `nil`は真か偽か？ コードを書いて答えを確かめてみてほしい．
- プロトタイプに存在するスロットを確認するにはどうすればよいか？
- `=`（等号），`:=`（コロン等号），および`::=`（コロンコロン等号）の違いは何か？ それぞれ，どのようなときに使うか？

試してみよう：

- ファイルからIoのプログラムを実行せよ．
- スロットの名前を指定して格納されているコードを実行せよ．

少し時間をとって，スロットとプロトタイプについていろいろ調べてみよう．プロトタイプの動作方法を確実に理解しておいてほしい．

3.3　2日目：ソーセージキング

フェリス・ビューラーのことを今一度思い出してほしい．映画では，中流階級の高校生であるフェリスが，一流のレストランでシカゴのソーセージキングだとはったりをかますシーンがある．型破りなことが大好きなフェリスはまんまと最高のテーブルに座ってしまう．Javaでの開発経験があり，Javaを気に入っている人は，あのシーンを見て「一体何が起こったんだ？ 自由奔放がいつもいいとは限らないだろう．フェリスなんて奴は店から放り出されて当然だ」と思っているだろう．しかしIoでは，あまり堅苦しく考えずに少しリラックスして，そのパワーを活用する必要がある．Perl育ちの人は，フェリスのはったりが気に入ったことだろう．はったりがばれずにうまくいってしまうからだ．ものごとに対して気ままで無頓着な人は，逆にIoでは少し控えて行儀良くする必要があるだろう．2日目の今日は，Ioのスロットとメッセージを使って言語の基本的な動作を覗いてみよう．

条件とループ

Ioの条件文はすべて，シンタックスシュガーなしで実装されている．分かりやすく覚えやすいと思える反面，シンタックスによる手がかりが少ないのでコードが少し読みづらくなる．単純な無限ループは簡単にできる．ループを抜けるにはControl+Cを押せばよい．

```
Io> loop("getting dizzy..." println)
getting dizzy...
getting dizzy...
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
...
getting dizzy.^C
IoVM:
        Received signal. Setting interrupt flag.
...
```

ループはさまざまな並行性構文に有用だが，普通は条件付きのループ構文を使うだろう．例えばwhileループがそのひとつだ．whileループは評価する条件とメッセージを引数にとる．セミコロンは2つの異なるメッセージを連結することに注意されたい．

```
Io> i := 1
==> 1
Io> while(i <= 11, i println; i = i + 1); "This one goes up to 11" println
1
2
...
10
11
This one goes up to 11
```

forループでも同じことを実行できる．forループは，カウンタ名，初期値，終了値，増分（オプション），およびsenderを伴うメッセージを引数にとる．

```
Io> for(i, 1, 11, i println); "This one goes up to 11" println
1
2
...
10
11
This one goes up to 11
==> This one goes up to 11
```

オプションの増分を指定した例を挙げる．

```
Io> for(i, 1, 11, 2, i println); "This one goes up to 11" println
1
3
5
7
9
11
This one goes up to 11
==> This one goes up to 11
```

実際には，任意の数の引数を指定できることが多い．オプションの引数は第3引数であることに気づいただろうか．Ioでは余分な引数を追加できる．これは便利なようだが，慎重に使う必要がある．細かいところまで面倒をみてくれるコンパイラは存在しないからだ．

```
Io> for(i, 1, 2, 1, i println, "extra argument")
1
2
==> 2
Io> for(i, 1, 2, i println, "extra argument")
2
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
==> extra argument
```

最初の形式では，“extra argument” はその名のとおり余分だ．2 番目の形式では，オプションの増分引数を省略したため，残りの引数が左側にシフトされている．このため，“extra argument” が今度はメッセージとなり，i println（i を返す）刻みのループを実行している．このようなコードが複雑なパッケージの奥深くに潜んでいたら，Io は黙っておかしな動作をする．ときとして長所には短所が伴う．Io は自由だが，その自由が怪我のもとになることがある．

if 構文は，関数として，if(condition, true code, false code) という形式で実装されている．この関数は，条件が真であれば true code を，偽であれば false code を実行する．

```
Io> if(true, "It is true.", "It is false.")
==> It is true.
Io> if(false) then("It is true") else("It is false")
==> nil
Io> if(false) then("It is true." println) else("It is false." println)
It is false.
==> nil
```

以上，少しページを割いて制御文について説明した．これらの制御文を使って独自の演算子を作ることができる．次に演算子について説明する．

演算子

オブジェクト指向言語と同様，多くのプロトタイプ言語では，演算子を実現するシンタックスシュガーが用意されている．具体的には，特殊な形式を持つ + や/などの特殊なメソッドだ．Io では次のように，演算子表を直接確認することができる．

```
Io> OperatorTable
==> OperatorTable_0x100296098:
Operators
  0   ? @ @@
  1   **
  2   % * /
  3   + -
  4   << >>
  5   < <= > >=
  6   != ==
  7   &
  8   ^
  9   |
  10  && and
  11  or ||
  12  ..
  13  %= &= *= += -= /= <<= >>= ^= |=
  14  return

Assign Operators
  ::= newSlot
  :=  setSlot
  =   updateSlot
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:36:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
To add a new operator: OperatorTable addOperator("+", 4)
    and implement the + message.
To add a new assign operator: OperatorTable addAssignOperator(
    "=", "updateSlot") and implement the updateSlot message.
```

代入は異なる種類の演算子であることが分かる．左側の番号は優先順位を表す．0 番目に近い引数が最初に連結される．+ が先，==が後に評価され，*が先，+ が後に評価される．これは期待どおりだ．優先順位を変更するには () を使う．ここで，排他的 or 演算子 xor を定義してみよう．xor は，ちょうど 1 つだけの引数が真ならば true を返し，それ以外の場合は false を返す．まず，演算子表に演算子を追加する．

```
Io> OperatorTable addOperator("xor", 11)
==> OperatorTable_0x100296098:
Operators
  ...
  10  && and
  11  or xor ||
  12  ..
  ...
```

新しい演算子 xor が正しい場所に追加されたことが分かる．次に，true および false に対する xor メソッドを実装する必要がある．

```
Io> true xor := method(bool, if(bool, false, true))
==> method(bool,
    if(bool, false, true)
)
Io> false xor := method(bool, if(bool, true, false))
==> method(bool,
    if(bool, true, false)
)
```

ここでは考え方を簡単にするため，総当たりで調べてみよう．定義した演算子は，次のとおり，こちらの予想どおりに動作する．

```
Io> true xor true
==> false
Io> true xor false
==> true
Io> false xor true
==> true
Io> false xor false
==> false
```

最終的に，true xor true は true xor(true) として評価される．演算子表の xor メソッドには，優先順位と簡素化されたシンタックスが定義される．

代入演算子は別の表に定義され，少し異なる動作をする．代入演算子は，メッセージとして機能する．具体例については，3.4「ドメイン固有言語」(p. 62)を参照してほしい．現時点で，演算子について言えるのはそれだけである．では次にメッセージの説明に移ろう．メッセージの項では，プログラマが自分で制御構造を実装する方法について学ぶ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

メッセージ

この章を執筆している最中，Io のコミッターのひとりが手伝ってくれたので，挫折しそうな時期を何とか乗り越えることができた．彼曰く「Bruce，Io に関して理解しておかなければならないことがあるんだ．Io では，ほとんどすべてのものはメッセージなんだよ」．Io のコードを見ると，コメントマーカーおよび引数間のカンマ（,）以外はすべてメッセージだ．Io を深く学ぶということは，すなわち基本的な呼び出しの域を超えたメッセージ操作について学ぶことである．Io の最も重要な機能の 1 つにメッセージのリフレクションがある．任意のメッセージの性質を問い合わせて，それに応じて処理を実行することができる．

メッセージは，sender（送信元），target（宛先），arguments（引数）の 3 つのコンポーネントからなる．Io では sender が target にメッセージを送信し，target がメッセージを実行する．

call メソッドを使うと，任意のメッセージに関するメタ情報を参照できる．ここで，いくつかオブジェクトを作成してみよう．メッセージを取得する postOffice オブジェクトと，メッセージを配信する mailer オブジェクトだ．

```
Io> postOffice := Object clone
==>  Object_0x100444b38:

Io> postOffice packageSender := method(call sender)
==> method(
    call sender
)
```

次に，メッセージを配信するメーラーを作成する．

```
Io> mailer := Object clone
==>  Object_0x1005bfda0:

Io> mailer deliver := method(postOffice packageSender)
==> method(
    postOffice packageSender
)
```

このオブジェクトにはスロットが 1 つある．deliver スロットだ．このスロットは，packageSender メッセージを postOffice に送信する．これで，mailer にメッセージを配信させることができる．

```
Io> mailer deliver
==>  Object_0x1005bfda0:
  deliver          = method(...)
```

つまり，deliver メソッドというオブジェクトがメッセージを送信したわけだ．次のように，宛先を取得することもできる．

```
Io> postOffice messageTarget := method(call target)
==> method(
    call target
)
Io> postOffice messageTarget
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
==>  Object_0x1004ce658:
  messageTarget    = method(...)
  packageSender    = method(...)
```

単純明快だ．宛先は，スロット名から分かるように郵便局（post office）だ．元のメッセージ名と引数は，次のようにして取得する．

```
Io> postOffice messageArgs := method(call message arguments)
==> method(
    call message arguments
)
Io> postOffice messageName := method(call message name)
==> method(
    call message name
)
Io> postOffice messageArgs("one", 2, :three)
==> list("one", 2, : three)
Io> postOffice messageName
==> messageName
```

このように Io には，メッセージのリフレクションを実行するために使えるメソッドが多数用意されている．次の疑問は，「Io はいつメッセージを評価するのか」という点だ．

ほとんどの言語では，引数をスタック値として渡す．例えば Java では，引数の各値を最初に計算してから，それらの値をスタックに積む．Io ではそのようなことはせず，メッセージ自体とそのコンテキストをそのまま渡す．そしてレシーバ側でそのメッセージを評価する．実際，制御構造はメッセージで実装することができる．Io の if 文を思い出してほしい．形式は，if(booleanExpression, trueBlock, falseBlock) だ．ここで unless を実装したいなら，次のように書けばよい．

▶ code/io/unless.io

```
unless := method(
    (call sender doMessage(call message argAt(0))) ifFalse(
    call sender doMessage(call message argAt(1))) ifTrue(
    call sender doMessage(call message argAt(2)))
)

unless(1 == 2, write("One is not two\n"), write("one is two\n"))
```

この例は簡単だが美しいので，注意深く読んでほしい．doMessage は Ruby の eval のようなものだが，eval よりもさらにローレベルで実行される．Ruby の eval は文字列をコードとして評価するが，doMessage は任意のメッセージを実行する．Io では，メッセージの引数を解釈するが，バインディングと実行は遅延させる．典型的なオブジェクト指向言語では，インタープリタまたはコンパイラが，コードブロックを含め，すべての引数を評価して，戻り値をスタックに積む．Io ではまったく様子が異なる．

例えば，オブジェクト westley がメッセージ princessButtercup unless(trueLove, ("It is false" println), ("It is true" println)) を送信する場合を考えてみる．メッセージセンドの一連の流れは次のようになる．

1. オブジェクト westley が上のメッセージを送信する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

2. Io が解釈されたメッセージとコンテキスト（call を用いて取得した送信元，宛先，メッセージ）を受け取り，スタックに積む．
3. princessButtercup がメッセージを評価する．unless スロットは存在しないため，Io はプロトタイプチェーンを上向きにたどって，unless を探す．
4. Io が unless メッセージの実行を開始する．まず，call sender doMessage(call message argAt(0)) を実行する．このコードは，westley trueLove というコードに単純化される．“The Princess Bride”[*5]という映画をご覧になった方ならお分かりだと思うが，westley には trueLove というスロットがあり，その値は true である．
5. このメッセージは偽ではないため，3 番目のコードブロックを実行する．3 番目のコードブロックは，westley ("It is true" println) というコードに単純化される．

ここでは，Io が unless 制御構文を実装するのに，引数を実行して戻り値を計算するという動作をしない，という事実を利用している．この仕組みは極めて強力である．ここまでは，リフレクションの一方の側面，すなわちメッセージリフレクションの動作について見てきた．リフレクションのもう一方の側面は，状態である．ここからはオブジェクトのスロットの状態について見ていこう．

リフレクション

Io には，スロットで起こっていることを理解するためのシンプルなメソッドがいくつか用意されている．次の例は，そのうちのいくつかを使ったものだ．このコードでは，数個のオブジェクトを作成し，ancestors というメソッドを使ってプロトタイプチェーンを上向きにさかのぼっている．

▶ code/io/animals.io

```
Object ancestors := method(
        prototype := self proto
        if(prototype != Object,
        writeln("Slots of ", prototype type, "\n---------------")
        prototype slotNames foreach(slotName, writeln(slotName))
        writeln
        prototype ancestors))


Animal := Object clone
Animal speak := method(
            "ambiguous animal noise" println)

Duck := Animal clone
Duck speak := method(
            "quack" println)
```

[*5] ［訳注］邦題は『プリンセス・ブライド・ストーリー』．美しいバターカップ姫と，彼女と真実の愛を誓った若者ウェスリーの物語．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Duck walk := method(
            "waddle" println)

disco := Duck clone
disco ancestors
```

このコードはそれほど複雑ではない．最初に，Animal プロトタイプを作成し，それを使って Duck インスタンスを作成し，speak メソッドを定義する．disco のプロトタイプは Duck である．ancestors メソッドはオブジェクトのプロトタイプのスロットを出力し，そのプロトタイプに対して ancestors を呼ぶ．注意してほしいのだが，1 つのオブジェクトが複数のプロトタイプを持つ可能性がある．ただ，ここではそうしたケースは取り上げない．紙面の都合上，再帰処理を停止しているので，Object プロトタイプのスロットは一部しか出力されていない．このコードを io animals.io として実行してみよう．

結果は次のようになる．

```
Slots of Duck
---------------
speak
walk
type

Slots of Animal
---------------
speak
type
```

驚くようなことは何もない．すべてのオブジェクトにはプロトタイプがあり，それらのプロトタイプもまた，スロットを持つオブジェクトである．Io でのリフレクションには 2 つの側面がある．前述の郵便局（post office）の例では，メッセージリフレクションについて見た．もう 1 つはオブジェクトリフレクションだ．オブジェクトリフレクションでは，オブジェクトとオブジェクトのスロットを扱う．クラスはどこにも登場しない．

2 日目で学んだこと

ここまでの説明についてきた方は，この 2 日目で，ある種のブレイクスルーを体験したに違いない．もうあなたは，ドキュメントの助けを少し借りるだけで基本的なタスクをこなせるくらいの Io の知識が身についているはずだ．条件文の書き方，メソッドの定義方法，データ構造体の使用方法，基本的な制御構文の使用方法について学んだ．この演習では，Io の能力を試してみる．存分に Io に慣れ親しんでほしい．メタプログラミングや並行性の領域に Io を拡張する際の問題に進むには，これらの基本事項を押さえておかなければならない．

セルフスタディ 2 日目

試してみよう：

1. フィボナッチ数列は 2 つの連続する 1 で始まる．以降，それぞれの数は直前の 2

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

つの数の和になっている．1, 1, 2, 3, 5, 8, 13, 21,... という具合だ．フィボナッチ数列の n 番目の数を見つけるプログラムを書け．例えば，`fib(1)` は 1，`fib(4)` は 3 となる．また，この問題を再帰とループを用いて解け．

2. 分母がゼロのときゼロを返すように/演算子を変更するには，どうすればよいか．
3. 2 次元配列のすべての要素の数値の総和を求めるプログラムを書け．
4. リストに，そのリストのすべての数値の平均を算出するスロット `myAverage` を追加せよ．リストに数値が含まれていない場合は何が起こるか？（おまけ：リスト内に数値以外の項目を見つけたら，Io 例外を発生させるようにしてみよ．）
5. 2 次元リストを実現するプロトタイプを書け．`dim(x, y)` メソッドは，要素数が x のリスト y のリストを割り当てる．`set(x, y, value)` はリストに値を設定し，`get(x, y)` はリストの値を取得する．
6. ボーナス問題：`(new_matrix get(y, x)) == matrix get(x, y)` が成立するように，元のリストの転置メソッドを書け．
7. 行列をファイルに書き出し，ファイルから行列を読み込め．
8. 1〜100 の数のなかからランダムに数を 1 つ取り出し，それを 10 回以内に当てさせるプログラムを書け．できれば，2 回目以降，前回より正解に近づいたか遠ざかったをヒントとして与えるようにしてみよ．

3.4　3日目：パレードとその他の奇妙な場所

Io を始めて数日間はフラストレーションが溜まったものの，数週間も経つと，予想もしなかったような使い方に気づき，思わず女子学生のようにくすくす笑いながらコーディングを楽しむようになった．これはちょうど，フェリスがニュース番組や野球場やパレードなど，こちらが想像もしないような場所に現れるのとよく似ている．最終的には自分の望みどおりに Io を操作できるようになった．Io は筆者の思考方法を変えてくれた言語だ．

ドメイン固有言語

Io をかなり使い込んだ経験のあるプログラマは例外なく，Io のドメイン固有言語（DSL）としての機能を高く評価する．Io のコアコミッターの一人である Jeremy Tregunna によると，C のサブセットを Io で実装したところ，わずか 40 行程度のコードで済んだという．この例は少し内容が深すぎるため，ここでは Jeremy が書いた別の素晴らしい作品を取り上げることにする．この作品は，電話番号を表現する面白い構文を提供する API を実装したものだ．

例えば，電話番号を次の形式で表現したいとする．

```
{
    "Bob Smith": "5195551212",
    "Mary Walsh": "4162223434"
}
```

このようなリストを管理する問題にはたくさんのアプローチがある．2 つほど思い浮かぶのは，リストを解析する方法とリストを解釈する方法だ．リストを解析する方法では，

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

さまざまな構文要素を認識するプログラムを書き，Io が理解する構造体に変換する．この方法は後の宿題としてとっておこう．それよりも，この電話番号一覧を Io のハッシュとして解釈したほうがずっと面白い．それには，Io を変更する必要がある．変更が済むと，Io はこのリストを，ハッシュを構築するための有効なシンタックスとして受け入れるようになる．

以下のコードは，Jeremy が Chris Kappler と協力してこの問題を解いたものだ．Chris はこのサンプルを最新バージョンの Io に合わせてアップデートしてくれた．

▶ code/io/phonebook.io

```
OperatorTable addAssignOperator(":", "atPutNumber")
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
)
Map atPutNumber := method(
  self atPut(
       call evalArgAt(0) asMutable removePrefix("\"") removeSuffix("\""),
       call evalArgAt(1))
)
s := File with("phonebook.txt") openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys   println
phoneNumbers values println
```

このコードはこれまで見てきたものに比べて少し複雑だが，基本的な構築ブロックについては分かるはずだ．では分解して説明していこう．

▶ code/io/phonebook.io

```
OperatorTable addAssignOperator(":", "atPutNumber")
```

1 行目では，Io の代入演算子テーブルに演算子を追加している．:は常に，atPutNumber として解析される．第 1 引数は名前（つまり文字列），第 2 引数は値として解釈される．したがって key : value は atPutNumber("key", value) のように解釈される．次に進もう．

▶ code/io/phonebook.io

```
curlyBrackets := method(
  r := Map clone
  call message arguments foreach(arg,
       r doMessage(arg)
       )
  r
)
```

パーサーは，中括弧（{}）に出会うたびに curlyBrackets メソッドを呼び出す．curlyBrackets メソッドでは空のマップを作成する．その後，各引数に対して call

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

message arguments foreach(arg, r doMessage(arg)) を実行する．この行は，多くの内容が圧縮された密度の高い行だ．分解して見ていこう．

左から右に順に見ていく．まず call message によって，中括弧で囲まれたコード（電話番号）が取得される．次にリスト内の各電話番号を forEach で回す．各名前と電話番号ごとに r doMessage(arg) を実行する．例えば最初の電話番号は r "Bob Smith": "5195551212"として実行される．:は atPutNumber として演算子テーブルに登録されているため，r atPutNumber("Bob Smith", "5195551212") が実行されることになる．次の部分に進もう．

▶ code/io/phonebook.io

```
Map atPutNumber := method(
  self atPut(
       call evalArgAt(0) asMutable removePrefix("\"") removeSuffix("\""),
       call evalArgAt(1))
)
```

key : value は atPutNumber("key", value) として解析されることを思い出してほしい．この例ではキーに既に文字列が入っているため，先頭と末尾の引用符を削除する．atPutNumber は，ターゲット範囲，つまり self に対して atPut を呼び出し，第 1 引数から引用符を削除しているだけである．メッセージは変更不能な値なので，引用符を削除するにはメッセージを変更可能な値に変換する必要がある．

このコードは，例えば次のように使う．

▶ code/io/phonebook.io

```
s := File with("phonebook.txt") openForReading contents
phoneNumbers := doString(s)
phoneNumbers keys   println
phoneNumbers values println
```

Io のシンタックスを理解するのは難しくない．ライブラリ内で何が行われているかが分かればよい．この例では，いくつか新しいライブラリが使われている．doString メッセージは，電話帳をコードとして評価する．File はファイルを操作するためのプロトタイプで，with にファイル名を指定すると，ファイルオブジェクトが返される．openForReading は，当該ファイルを開き，ファイルオブジェクトを返す．contents は当該ファイルの内容を返す．以上をつなげると，電話帳を読み込み，コードとして評価するコードとなる．

次に中括弧によってマップが定義される．マップ"string1" : "string2"内の各行により，map atPut("string1", "string2") が実行され，電話番号のハッシュが出来上がる．Io ではこのように，演算子からシンボルまで言語を構成するすべてのものを再定義できるため，自分の思いどおりに DSL を構築できる．

以上の説明で，Io のシンタックスを変更する方法を理解できるようになったはずだ．さて，Io の動作を動的に変更するにはどうすればよいだろうか．これが次のセクションのトピックだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Io の method_missing

制御フローについて復習しておこう．与えられたメッセージに対する動作は，Object 内にすべて定義済みである．このオブジェクトにメッセージを送信すると，次の処理が実行される．

1. 引数をすべて計算する．引数は単なるメッセージである．
2. メッセージの名前，宛先，送信元を取得する．
3. ターゲットのメッセージの名前でスロットを読み込む．
4. スロットが存在していれば，データを返すか，内部でメソッドを呼び出す．
5. スロットが存在しない場合は，メッセージをプロトタイプに転送する．

これらは Io 内で継承を実現する基本的なメカニズムであり，通常は変更する必要はない．

しかし変更することは可能だ．forward メッセージは Ruby の method_missing と同じように使えるが，少しリスクが高い．Io にはクラスがないため，forward を変更すると，object に定義されている基本的な振る舞いを取得する方法も変更されてしまう．言ってみれば，綱渡りをしながら手斧を曲投げするようなものだ．とはいえ，うまく使えば気の利いた処理ができる．では始めよう．

XML は，シンタックスは美しくないが，データを見事に構造化してくれる．XML データを Io のコードとして表現する仕組みを構築すれば役に立つだろう．

例えば次のような XML データがあるとする．

```
<body>
<p>
This is a simple paragraph.
</p>
</body>
```

これを次のように書けるようにしたい．

```
body(
    p("This is a simple paragraph.")
)
```

この新しい言語を LispML と呼ぶことにしよう．Io の forward を method_missing と同様に利用する．以下にコードを示す．

▶ code/io/builder.io

```
Builder := Object clone

Builder forward := method(
  writeln("<", call message name, ">")
  call message arguments foreach(
        arg,
        content := self doMessage(arg);
        if(content type == "Sequence", writeln(content)))
  writeln("</", call message name, ">"))

Builder  ul(
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
        li("Io"),
        li("Lua"),
        li("JavaScript"))
```

では解説しよう．この Builder プロトタイプは非常に便利だ．forward をオーバーライドして，任意のメソッドを選択する．最初に，開始タグを出力する．次にちょっとしたメッセージ・リフレクションを用いる．すなわち，メッセージが文字列なら，Io はそれをシーケンスとして認識し，Builder がその文字列をクオートなしで出力する．最後に終了タグを出力する．

次のように予想どおりの結果が得られる．

```
<ul>
<li>
Io
</li>
<li>
Lua
</li>
<li>
JavaScript
</li>
</ul>
```

LispML が従来の XML と比較して格段に進歩しているかどうか自信はないが，この例は示唆に富んでいる．Io のプロトタイプの 1 つで継承が行われる方法を完全に変えてしまったのだ．Builder のインスタンスはすべて同じように振る舞う．これにより，Io のシンタックスを持つものの，振る舞いは Io とはまったく異なる新しい言語を作成できる．具体的には，独自の Object を定義し，すべてのプロトタイプをそのオブジェクトに基づいて作成する．Object をオーバーライドして新しいオブジェクトを複製（clone）することさえ可能だ．

並行性

Io には優れた並行性ライブラリが用意されている．主なコンポーネントはコルーチン，アクター，フューチャの 3 つだ．

コルーチン

並行性の基盤はコルーチンだ．コルーチンを使うと，プロセスの実行を自由に停止したり再開したりできる．複数の入口点と出口点を持つ関数のようなものだと考えてよい．yield を呼び出すたびに，プロセスが停止され，別のプロセスに制御が移る．メッセージの前に@または@@を指定することで，メッセージを非同期に送信することもできる．前者はフューチャ（後述）を返す．後者は nil を返し，自身のスレッドでメッセージを開始する．次のプログラムで考えてみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:38:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/io/coroutine.io

```
vizzini := Object clone
vizzini talk := method(
            "Fezzik, are there rocks ahead?" println
            yield
            "No more rhymes now, I mean it." println
             yield)

fezzik := Object clone

fezzik rhyme := method(
                      yield
            "If there are, we'll all be dead." println
            yield
            "Anybody want a peanut?" println)

vizzini @@talk; fezzik @@rhyme

Coroutine currentCoroutine pause
```

fezzik と vizzini は，Object の個別のインスタンスで，どちらもコルーチンを使っている．talk メソッドと rhyme メソッドは非同期に呼び出される．この 2 つのメソッドは，yield メッセージを用いて，指定された間隔で互いに相手に制御を自発的に譲りながら並列に動作する．最後の pause によって，すべての非同期メッセージが完了するまで待ち，終了する．コルーチンは，協調的なマルチタスクを必要とするソリューションにうってつけだ．この例では，協調する必要がある 2 つのプロセスがいとも簡単に協調して，次のような詩が出来上がる．

```
batate$ io code/io/coroutine.io
Fezzik, are there rocks ahead?
If there are, we'll all be dead.
No more rhymes now, I mean it.
Anybody want a peanut?
Scheduler: nothing left to resume so we are exiting
```

Java と C ベースの言語は，プリエンプティブマルチタスキングと呼ばれる並行性の概念を用いている．この並行方式と変更可能な状態を持つオブジェクトを組み合わせると，予測が難しく，ほとんどの開発チームで採用している現在のテスト方法ではデバッグがほぼ不可能なプログラムになってしまう．コルーチンはこうした従来の並行性とは異なる．コルーチンを使うと，アプリケーションは適当なタイミングで自ら制御を譲ることができる．例えば分散クライアントは，サーバを待つ間は制御を放棄することができる．ワーカープロセスはキュー内のアイテムを処理した後，一時停止できる．

コルーチンは，アクターのような高レベルの抽象化を実現する基本的な構築ブロックである．アクターは，メッセージの送信，メッセージの処理，他のアクターの作成を行う汎用の並行性プリミティブのようなものだと考えてほしい．アクターは複数のメッセージを同時に受信する．Io では，アクターは受信したメッセージをキューに登録し，キューの内容をコルーチンで処理する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

次の項ではアクターについて説明する．アクターを使ったコーディングは驚くほど簡単なので信じられないかもしれない．

アクター

アクターは，理論上はスレッドよりもはるかに有利だ．アクターは，厳密に制御されたキューを介してのみ，自分の状態を変更したり他のアクターにアクセスしたりできる．スレッドは，互いの状態を制限なしに自由に変更できる．このためスレッドでは競合状態と呼ばれる並行性の問題が発生する．競合状態が発生すると，2つのスレッドが同時にリソースにアクセスし，予測不能な結果になる．

Io の良いところは，非同期のメッセージをオブジェクトに送信すると，そのオブジェクトがアクターになる点だ．簡単な例で見てみよう．まず2つのオブジェクト faster と slower を作る．

```
Io> slower := Object clone
==>  Object_0x1004ebb18:

Io> faster := Object clone
==>  Object_0x100340b10:
```

次に，各オブジェクトに start というメソッドを追加する．

```
Io> slower start := method(wait(2); writeln("slowly"))
==> method(
    wait(2); writeln("slowly")
)
Io> faster start := method(wait(1); writeln("quickly"))
==> method(
    wait(1); writeln("quickly")
)
```

これで，2つのメソッドを1行のコードで順番に呼び出して，簡単なメッセージを送信できる．

```
Io> slower start; faster start
slowly
quickly
==> nil
```

2つのメソッドは順番に開始される．最初のメッセージが終了するまで，次のメッセージは開始できないからだ．しかし，各オブジェクトをそれぞれ別のスレッドで実行させるのは簡単だ．各メッセージの前に@@を付ければよい．これで，メッセージ自体は即座に復帰し nil を返す．

```
Io> slower @@start; faster @@start; wait(3)
quickly
slowly
```

最後に余分な待ち時間を追加したのは，すべてのスレッドが完了してからプログラムを終了させるためだ．予想どおりの結果だ．2つのスレッドで動作するようになった．2つのオブジェクトをアクターにしたわけだが，やったことといえば，2つのオブジェクト

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

に非同期メッセージを送信しただけである．

フューチャ

並行性に関するトピックの締めくくりとして，フューチャの概念について説明する．フューチャ（future）とは，非同期のメッセージ呼び出しによって即座に返される結果オブジェクトのことだ．メッセージの処理には少し時間がかかるため，プロセスの結果が戻ってきた時点でフューチャが本当の結果となる．結果が戻ってくる前にフューチャの値を尋ねると，そのプロセスは結果値が使用可能になるまでブロックされる．例えば，次のように実行に時間のかかるメソッドがあるとする．

```
futureResult := URL with("http://google.com/") @fetch
```

メソッドを実行したあと，即座に何か別の処理を開始し，結果が戻ってくるまで待つことができる．

```
writeln("Do something immediately while fetch goes on in background...")
// …
```

プロセスの結果が戻ってきたら，フューチャの値を使うことができる．

```
writeln("This will block until the result is available.")
// この行は即座に実行される

writeln("fetched ", futureResult size, " bytes")
// プロセスの処理が終了するまでこの行はブロックされる

// Io が値を出力する
==> 1955
```

上記のコード futureResult は，フューチャオブジェクトを即座に返す．Io では，フューチャはプロキシではない[*6]．フューチャは評価が終了し結果オブジェクトが得られるまでブロックされる．結果が戻ってくるまでは値は Future オブジェクトであり，結果が戻ってくると値がすべて結果オブジェクトを指すように置き換えられる．コンソールには，返された最後の文の文字列値が出力される．

Io のフューチャには，デッドロックを自動的に検出する機能も備えている．なかなか気の利いた機能で，理解も容易で使いやすい．

以上で，Io の並行性機能の感じがつかめたことと思う．これで，Io を評価するためのしっかりした基礎は習得できた．覚えたことを実際に使えるように，3 日目の内容をまとめておこう．

3 日目で学んだこと

3 日目は，Io の少し重要な機能について学んだ．まず，シンタックスを変更し，ブレースを用いてハッシュを作る新しいシンタックスを構築した．演算子表に演算子を追加して，ハッシュテーブルを操作できるようにした．また，method_missing を用いて XML

[*6] ［訳注］多くの言語ではフューチャとしてプロキシ，つまり未来のオブジェクトをラップするオブジェクトを返すが，Io ではフューチャで返された値が本物の値に「すりかわる」．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

の要素を出力する XML ジェネレータを構築した．

次に，コルーチンを使って並行性を管理するコードを書いた．コルーチンは，Ruby や C や Java といった言語の並行性とは異なる．スレッドは自分自身の状態しか変更できないため，予測可能性が高く分かりやすいモデルとなっており，ボトルネックとなる状態のブロックが必要となるケースも少ない．

いくつかの非同期メッセージを送信して，プロトタイプをアクターにした．メッセージのシンタックスを変更するだけで，ほかには何もする必要がなかった．最後に，フューチャと Io でのフューチャの動作方法について簡単に説明した．

セルフスタディ 3 日目

試してみよう：

- 本文に掲載した XML プログラムを改良して，インデントを示すスペースを追加するようにしてみよ．
- 角括弧を使ったリスト構文を作成せよ．
- 本文に掲載した XML プログラムを改良して属性を処理できるようにしてみよ．具体的には，第 1 引数がマップ（丸括弧構文を使用）の場合は，それらの各属性を XML プログラムに追加する．次に例を示す．
  `book({"author": "Tate"}...)` と書くと，`<book author="Tate">` と出力される

3.5　Io のまとめ

Io はプロトタイプベースの言語の使い方を学ぶにはうってつけの言語だ．シンタックスは，Lisp 同様，極めて簡単だが，言語のセマンティクスは大変な力を実現している．プロトタイプ言語でも，オブジェクト指向言語同様，データと振る舞いをカプセル化する．継承はオブジェクト指向言語よりも単純だ．Io にはクラスやモジュールは存在しない．オブジェクトは，そのプロトタイプから直接振る舞いを継承する．

長所

プロトタイプ言語は一般に，極めて柔軟性が高い．どのオブジェクトのどのスロットも自由に変更できる．Io ではこの柔軟性を最大限に活かして，プログラマが必要なシンタックスをすぐに作成できる．Ruby と同様に Io には，動的な性質を実現した代償として，少なくとも単一スレッドではパフォーマンスが制限される傾向がある．また，高機能の最新ライブラリのおかげで，Io は並列処理に向いた言語になっている．以下，Io が優れている点について 1 つずつ見ていこう．

所要メモリ

Io は小さなメモリ領域で動作する．本番環境で動作する Io アプリケーションはほとんど組み込みシステムだ．Io が，コンパクトでパワフル，かつ柔軟性の高い言語であることを考えると，それもうなずける．仮想マシンは別のオペレーティング環境に容易に移

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

植できる．

シンプルさ

Io のシンタックスは驚くほどコンパクトだ．だからすぐに覚えられる．主要なシンタックスを理解してしまえば，あとはライブラリ構造を覚えるだけだ．筆者は，Io を始めて 1 か月以内という極めて短期間でメタプログラミングまでできるようになった．Ruby でメタプログラミングができるようになるには，もう少しかかった．Java にいたっては，メタプログラミングというものが少しでも分かるレベルに到達するまでに何ヶ月もかかった．

柔軟性

Io のダックタイピングと自由さのおかげで，オブジェクトの任意のスロットをいつでも自由に変更できる．この自由奔放さは極めて高く，言語の基本的なルールさえ自分のアプリケーションに合わせて変更できる．forward スロットを変更することによって，どこにでもごく簡単にプロキシを追加できる．スロットを直接変更することによって言語の主要構文をオーバーライドすることもできる．独自のシンタックスでさえ簡単に作成できる．

並行性

Java や Ruby と違って，Io の並行性構文は最新の概念を導入している．アクター，フューチャ，コルーチンを使うことで，マルチスレッドアプリケーションを従来よりもはるかに容易に構築できる．しかも従来に比べて，テストが容易でパフォーマンスも高い．また Io では，変更可能なデータの扱い方とその回避方法がよく練られている．これらの機能がコアライブラリに組み込まれているため，堅牢な並行性モデルを簡単に覚えることができる．これらの概念については，後出の他の言語の章でも説明する．アクターは Scala や Erlang でも登場する概念だ．

短所

Io には好ましい性質もたくさんあるが，同時に，最善とは言えない側面もいくつかある．自由さと柔軟性には犠牲が伴う．また，Io のコミュニティは，本書で紹介した言語のなかでは最も規模が小さいため，プロジェクトによっては Io を選択するのはリスクが大きい．では Io の問題点について見ていこう．

シンタックス

Io にはシンタックスシュガーがほとんどない．簡潔なシンタックスは諸刃の剣だ．明解なシンタックスは言語を分かりやすくするが，それには犠牲が伴う．シンタックスが簡潔だと難しい概念をコンパクトに伝えるのが困難になることが多い．言い換えると，あるプログラムが Io を使っている理由は簡単に分かっても，そのプログラムの処理内容を理解するのは容易ではない．

この点を Ruby と比較してみよう．まず，`array[-1]` という Ruby のコードは，`-1` が配列の最後の要素であることを知らなければ面食らうかもしれない．また，`[]` が，配列

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

の指定されたインデックスの値を取得するメソッドであることも覚えておく必要がある．こうした概念を理解してはじめて，より多くのコードを一目見ただけで理解できるようになる．Io には，これと正反対のことが当てはまる．つまり，初心者が覚える必要のあることは少なくて済むが，シンタックスシュガーを使えば簡単に伝えることができる概念でも，それがないばかりに習得するのに少しばかり苦労させられることになる．

シンタックスシュガーをどの程度組み込むかは難しい問題だ．シンタックスシュガーを多用すると，難しすぎて言語とその使い方を覚えられない．逆にシンタックスシュガーが少なすぎると，コーディングの所要時間が長くなる．また，デバッグにはさらにエネルギーが必要になる可能性がある．結局，シンタックスは好みの問題だ．Matz はシンタックスシュガーを多用することを選択した．Steve はそうしなかった，ということだ．

コミュニティ

現時点では，Io のコミュニティは小規模だ．Io では，他の言語と違って，目的のライブラリをいつも簡単に見つけられるとは限らない．Io のプログラマ自体も他の言語に比べて少ない．こうした欠点は，C の良いインタフェースを備えていること（さまざまな言語とやり取りできる）と，非常に覚えやすいシンタックスによっていくらかは軽減されている．とはいうものの，コミュニティの規模が小さいことは間違いなく弱点であり，強力な新しい言語がなかなか普及しない主な原因でもある．Io の場合も，キラーアプリケーションが登場して広く普及するか，このまま少数派言語に甘んじるかのどちらかになるだろう．

パフォーマンス

並行性やアプリケーション設計といった他の問題を抜きにして，パフォーマンスのことを議論するのはあまり賢いとは言えないが，Io には，シングルスレッドの実行速度を低下させるような性質が数多く存在することを指摘しておかなければならない．この欠点は Io の並列性構文によっていくらか軽減されるものの，やはり頭に入れておかなければならない．

終わりに

筆者は全般に Io を楽しんで覚えることができた．シンタックスが単純であることと実行に必要なメモリ容量も少なくて済むことに興味を引かれた．また Io では，Lisp と同様，単純さと柔軟性を最優先する考え方が徹底している．Steve Dekorte は，言語の開発時にこの考え方を一貫して尊重することで，Lisp のプロトタイプ言語版ともいえる言語を開発した．筆者は，Io は成長の方向へ向かっていると思う．フェリス・ビューラーのように，Io の未来は明るくもあり，危うくもある．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第 4 章

Prolog

Sally Dibbs, Dibbs Sally. 461-0192.
レイモンド

Prolog の登場だ．この言語は驚くほど利口な場合もあれば，それと同じくらいいらいらさせられることもある．素晴らしい答えが得られるのは，質問の仕方が分かっている場合だけだ．『レインマン』を思い浮かべてほしい[*1]．映画の中で，前の晩に電話帳を読んでいた主人公レイモンドが，Sally Dibbs の電話番号を無意識に早口で言うシーンがある．筆者は，レイモンドと Prolog のどちらについても，「なぜ分かったのか」と「なぜ分からないのか」という疑問を同じくらいの割合で抱くことがある．どちらも知識の泉である．ただし，正しい方法で質問をした場合に限る．

Prolog はこれまでに紹介した言語とは著しく異なる側面を持つ．Io と Ruby はどちらも命令型言語だ．命令型言語は目的を達成するための方法を記述する．コンピュータに仕事を行う方法を逐一教えてやるわけだ．低レベルの命令型言語よりも高レベルの命令型言語のほうが，より多くの手順をまとめられるので多少有利ではあるが，ケーキを焼くのに材料のリストを作って，手順を順番に記述する，という点ではどちらも基本的には同じだ．

この章を書くには，Prolog を数週間いじってみる必要があった．段階に応じてさまざまなチュートリアルを読んだ．例えば J. R. Fisher が書いたチュートリアル[*2]に掲載されているサンプルをじっくり研究したり，A. Aaby が書いた入門記事[*3]で言語の構造と用語を覚えたり，他にもさまざまな実験を行ったりした．

Prolog は宣言型言語である．Prolog で事実とルールを記述すると，Prolog が推論してくれる．言ってみれば，（自分でケーキを焼く代わりに）おいしいケーキ屋さんに行くようなものだ．食べたいケーキの特徴を記述すると，与えられたルールに基づいて，ケーキ屋さんが材料を選び，ケーキを焼いてくれる．Prolog では方法を知る必要はない．コンピュータがあなたに代わって推論してくれる．

インターネットで軽く検索してみると，数独を 20 行に満たないコードで解いたり，ルービックキューブを完成させたり，ハノイの塔などの有名なパズルを十数行程度のコードで解いたりしている例が見つかる．Prolog は初めて成功を収めた論理プログラミング言語のひとつだ．純粋論理を用いて表明を記述すると，それが真かどうかを Prolog が判

*1 『レインマン』．DVD，Barry Levinson 監督作品，1988 年．MGM（カリフォルニア州ロサンゼルス）制作，2000 年．

*2 http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html

*3 http://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

断してくれる．表明間に論理の飛躍があると，Prolog は，不完全な事実関係が矛盾しないように穴を埋めようとする．

4.1　Prolog について

Prolog は，Alain Colmerauer 氏と Phillipe Roussel 氏によって 1972 年に開発された論理プログラミング言語で，自然言語処理の世界で広く使われている．この由緒正しい言語は，スケジューリングからエキスパートシステムまで，広範な問題をプログラミングで解くための礎となった．ルールベース言語である Prolog を使えば，論理を表現したり質問をしたりできる．SQL と同様に Prolog はデータベースを扱うが，データは論理ルールと関係から成る．また，やはり SQL と同様に 2 つの部分，すなわちデータを表現する部分とデータに質問する部分で構成される．Prolog では，データは論理ルール形式で記述する．以下に Prolog の構成要素を示す．

- 事実．特定の世界についての基本的な表明（「ベーブは豚である」，「豚は泥が好きである」など）
- ルール．その世界の事実に関する推論（「動物が豚であれば，その動物は泥が好きである」など）
- 質問．その世界に関する質問（「ベーブは泥が好きか？」など）．

事実とルールは知識ベースを形成する．Prolog のコンパイラは，質問が効率的に処理できる形式に知識ベースをコンパイルする．この章の例では，Prolog を使って知識ベースを表現した後，データを直接取り出す．また，Prolog を使ってルールをつなぎ合わせ，質問者も思いつかないような答えを返させたりする．

前置きはこのくらいにして，さっそく始めよう．

4.2　1 日目：素晴らしいドライバー

映画『レインマン』に，レイモンドが弟に自分は運転がうまいと言う場面がある．といっても駐車場で時速 5 マイルで車を動かすくらいならうまくできるという意味でだ．確かに，ハンドル，ブレーキ，アクセルなど，主な要素（道具）はすべて使っている．ただ，それを使う状況が限られているだけだ．1 日目の目標はまさにこれだ．Prolog を使って事実を述べ，ルールを作成し，基本的な質問をする．Io と同様，Prolog はシンタックスという点では極めてシンプルな言語だ．構文規則だけならすぐに覚えられる．でも本当のお楽しみは，いろいろな概念をうまい具合に積み重ねていけるようになったときにやってくる．Prolog に初めて触れる人は，考え方が変わるほど大きな影響を受けるか，挫折してものにできずに終わるかのどちらかだ．詳細な文法については 2 日目以降で説明する．

まずは大事なことから．きちんと動作するバージョンを取得すること．本書では GNU Prolog 1.3.1 を使う．Prolog にはいくつも方言があり，記法などに差があるので注意してほしい．本書では，どの方言でも変わらない共通の部分に基づいて説明するように最善を尽くすが，GNU Prolog 以外を使う場合は，GNU Prolog と異なる部分を把握しておく必要がある．どの版を使うにしても，基本的な使い方は次のとおりだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

基本事実

言語によっては，大文字をどのように使うかはプログラマに一任されているが，Prologでは，先頭文字を大文字にするか小文字にするかは重要だ．小文字で始まる単語はアトム，すなわち，Rubyのシンボルのような固定値だ．大文字またはアンダースコアで始まる単語は変数である．変数の値は変更できるが，アトムは変更できない．では，数個の事実を用いて簡単な知識ベースを作成してみよう．次のようにエディタに入力してほしい．

▶ code/prolog/friends.pl

```
likes(wallace, cheese).
likes(grommit, cheese).
likes(wendolene, sheep).

friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

このファイルは事実とルールが定義された知識ベースだ．最初の3つの文が事実，最後の文がルールである．ここでいう事実とは現実世界をそのまま写し取ったものだ．ルールは現実世界に関する論理的な推論である．まずは最初の3行に注目してほしい．これらの各行は事実だ．wallace，grommit，およびwendoleneはアトムである．wallaceはcheeseが好きだ，grommitはcheeseが好きだ，wendoleneはsheepが好きだ，という具合に読む．これらの事実を実際にPrologに与えてみよう．

Prologインタープリタを起動する．GNU Prologならgprologと入力すればよい．次に，作成したファイルを読み込むために，次のように入力する．

```
| ?- ['friends.pl'].
compiling /Users/batate/prag/Book/code/prolog/friends.pl for byte code...
/Users/batate/prag/Book/code/prolog/friends.pl compiled, 4 lines read -
997 bytes written, 11 ms

yes
| ?-
```

Prologは，結果をまだすべて返していないために保留しているときを除き，yesまたはnoを返す．このケースでは，ファイルが正常に読み込まれたので，yesを返している．これで質問を始めることができる．最も基本的な質問は，事実に関してyesかnoかを尋ねる質問だ．いくつか質問してみよう．

```
| ?- likes(wallace, sheep).

no
| ?- likes(grommit, cheese).

yes
```

これらの質問は直感的だ．wallaceはsheepが好きか？（No）grommitはcheeseが好きか？（Yes）これではあまり面白くない．Prologは事実をオウム返ししているだけだからだ．何らかのロジックを組み込むようになれば，少し面白くなってくる．では推論の様子を少し見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

基本的な推論と変数

friend ルールを試してみよう．

```
| ?- friend(wallace, wallace).

no
```

Prolog は定義したルールを探索し，yes か no かを尋ねる質問に答える．ここは少し説明が必要だろう．friend ルールをもう一度見てほしい．

文章にすると，「X が Y の友人であるためには，X と Y は同一人物であってはならない」となる．:-の右側の最初の部分を見てほしい（この右側の部分をサブゴールという）．\+ は論理否定なので，\+(X = Y) は，X と Y は等しくないという意味になる．

もう少し質問してみよう．

```
| ?- friend(grommit, wallace).

yes
| ?- friend(wallace, grommit).

yes
```

文章にすると，「X が Z を好きで，Y もやはり Z が好きであるとき，X は Y の友人である」となる．wallace と grommit はどちらも cheese が好きなので，上の 2 つの質問に対する答えは yes となる．

コードを見てみよう．上の 2 つの質問では X と Y は等しくないので，最初のサブゴールは満たしている．次に，2 番目と 3 番目のサブゴール likes(X, Z) と likes(Y, Z) を見る．grommit と wallace は cheese が好きなので，2 番目と 3 番目のサブゴールも満たされる．別の質問をしてみよう．

```
| ?- friend(wendolene, grommit).

no
```

この場合，Prolog は X，Y，および Z について複数の値を試してみる必要がある．

- wendolene，grommit，cheese
- wendolene，grommit，sheep

どちらの組み合わせも両方のゴール，すなわち，「wendolene は Z が好きである」と「grommit は Z が好きである」を満たすことがないので，質問を満たすルールは存在しない．したがって論理エンジンは no を返す．すなわち，2 人は友人ではない．

Prolog 用語を使って説明してみよう．

```
friend(X, Y) :- \+(X = Y), likes(X, Z), likes(Y, Z).
```

これは 3 つの変数 X，Y，Z を使った Prolog のルールだ．このルールを friend/2 と呼ぶ．2 つのパラメータを持つ friend という意味の略称だ．このルールには 3 つのサブゴールがある．各サブゴールはカンマで区切って指定する．このルールが真であるためには，すべてのサブゴールが真でなければならない．したがって，このルールは「X と Y

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:40:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

が等しくなく，X と Y が共に Z を好きであるとき，X は Y の友人である」という意味になる．

穴埋め問題

ここまでは，Prolog を使って yes か no かの質問に答えてきたが，Prolog ではもっと複雑なことができる．ここでは論理エンジンを使って，質問に対して考えられるすべての答え（マッチ）を見つけてみる．それには，質問の中に変数を指定する．

次の知識ベースで考えてみよう．

▶ code/prolog/food.pl

```
food_type(velveeta, cheese).
food_type(ritz, cracker).
food_type(spam, meat).
food_type(sausage, meat).
food_type(jolt, soda).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, meat).
flavor(savory, cheese).
flavor(sweet, soda).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).
```

いくつか事実が定義されている．food_type(food, type) は，food が特定の type であるという意味だ．flavor(flavor, type) は，タイプが type の食品の風味は flavor であるという意味だ．最後に food_flavor というルールが定義されている．これは食品の風味を推論するルールだ．すなわち，食品 X が food_type Z（タイプ Z）で，その Z の風味が Y であるとき，食品 X は food_flavor Y（風味 Y）を持つ．コンパイルしてみよう．

```
| ?- ['code/prolog/food.pl'].
compiling /Users/batate/prag/Book/code/prolog/food.pl for byte code...
/Users/batate/prag/Book/code/prolog/food.pl compiled,
12 lines read - 1557 bytes written, 15 ms

(1 ms) yes
```

いくつか質問してみる．

```
| ?- food_type(What, meat).

What = spam ? ;

What = sausage ? ;

no
```

面白くなってきた．上の例では，Prolog に「food_type(What, meat) を満たす What の値を見つけよ」と尋ねている．1 つ見つかる．spam だ．; を入力すると，「さらに別の

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

値を見つけよ」という意味になる．今度は sausage が返される．質問が基本的な事実に基づいているため，これらの値を見つけるのは簡単だ．さらに別の値がないか尋ねると，no が返される．この動作は少し一貫性に欠けることがある．Prolog は，該当する値がもう存在しないことを検出すると，便宜上 yes を返す．さらに推論しないと別の該当値が存在するかどうかすぐには決められない場合は，次の質問を入力するよう促してから，no を返す．この機能は本当に便利だ．予想より早めに答えを返すことができる場合は，そうする．もう少し質問を続けよう．

```
| ?- food_flavor(sausage, sweet).

no

| ?- flavor(sweet, What).

What = dessert ? ;

What = soda

yes
```

sausage は sweet ではない．甘い食品のタイプは，dessert と soda だ．これらはすべて事実だが，Prolog に推論させることもできる．

```
| ?- food_flavor(What, savory).

What = velveeta ? ;

What = spam ? ;

What = sausage ? ;

no
```

food_flavor(X, Y) はルールであって事実ではない点に注意してほしい．つまりこの質問は「塩辛い風味（savory）の食品をすべて見つけよ」と Prolog に命令している．Prolog は，食品，タイプ，風味に関する基本的な事実を組み合わせて結論を導き出さなければならない．論理エンジンはすべてのゴールを真にする組み合わせをすべて調べる必要がある．

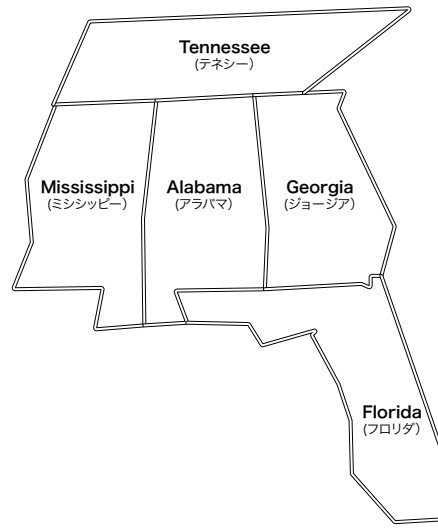
地図の塗り分け

同じ要領で地図の塗り分けを行ってみよう．Prolog の面目躍如たる一面を見てもらうために，米国南西部の地図を塗り分ける例を考えてみたい．対象となる州を図 4.1「米国南東部の州の地図」(次ページ)に示す．隣接する州を同じ色に塗らないようにしたい．

この単純な事実関係をコーディングする．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲ 図 4.1 米国南東部の州の地図

▶ code/prolog/map.pl

```
different(red, green). different(red, blue).
different(green, red). different(green, blue).
different(blue, red). different(blue, green).

coloring(Alabama, Mississippi, Georgia, Tennessee, Florida) :-
  different(Mississippi, Tennessee),
  different(Mississippi, Alabama),
  different(Alabama, Tennessee),
  different(Alabama, Mississippi),
  different(Alabama, Georgia),
  different(Alabama, Florida),
  different(Georgia, Florida),
  different(Georgia, Tennessee).
```

使うのは 3 色だ．最初に，塗り分けに使う異なる 2 色の組み合わせを指定する．次にルールを定義する．coloring ルールでは，隣接する州を指定する．コーディングは以上だ．動かしてみよう．

```
| ?- coloring(Alabama, Mississippi, Georgia, Tennessee, Florida).

Alabama = blue
Florida = green
Georgia = red
Mississippi = red
Tennessee = green ?
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

この5つの州を3色で塗り分ける方法が1つ見つかった．他の組み合わせがないか確認するには，a と入力する．わずか12行のコードで答えが見つかった．ロジックはこっけいなくらいシンプルだ．子供でも分かる．ぜひ一度，頭の中で考えてみてほしい．

プログラムはどこに？

なんと，アルゴリズムなんか気にしなくていいのだ．上の問題をお好みの手続き型言語で解いてみてほしい．解法は分かりやすいだろうか？ このような複雑なロジックを含む問題を Ruby や Io で解くための手順を考えてみるとよい．例えば次のような手順が考えられる．

1. ロジックをまとめて整理する．
2. ロジックをプログラムで表現する．
3. すべての解法を見つける．
4. 解法をプログラムで実装する．

この手のプログラムを何度もそのつど書くことになるだろう．Prolog では，事実と推論でロジックを表現し，利用者に質問させる．手順を追った解法を作る必要はない．Prolog では，論理的な問題を解くためにアルゴリズムを書くことはしない．現実世界をそのまま記述して，コンピュータが取り組めるような論理的な問題として提示する．

コンピュータに仕事をさせよう!

ユニフィケーションその1

ここで少し後戻りして，少し理論的な説明を行っておく．ユニフィケーションにもう少し焦点を当ててみたい．一部の言語では，変数への代入を行う．例えば Java や Ruby では，`x = 10` は，変数 `x` に `10` を代入するという意味だ．2つの構造体のユニフィケーションでは，両方の構造体を一致させようとする．次の知識ベースで考えてみよう．

▶ code/prolog/ohmy.pl

```
cat(lion).
cat(tiger).

dorothy(X, Y, Z) :- X = lion, Y = tiger, Z = bear.
twin_cats(X, Y) :- cat(X), cat(Y).
```

この例の=はユニファイ（unify），すなわち両辺を一致させるという意味だ．事実は「ライオンは猫である」と「トラは猫である」の2つ．ルールも簡単なものが2つ．`dorothy/3` では，`X`，`Y`，`Z` がそれぞれ `lion`，`tiger`，`bear`．`twin_cats/2` では，`X` は `cat`，`Y` は `cat` だ．この知識ベースを使って，ユニフィケーションについて簡単に説明してみる．

まず，最初のルールを使う．これをコンパイルして，パラメータなしの簡単な質問をする．

```
| ?- dorothy(lion, tiger, bear).

yes
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ユニフィケーションとは，「両辺を一致させるような値を見つけよ」という意味だった．右側では，Prologは，X，Y，Zをlion，tiger，bearにバインドする．これらのマッチは左側の対応する値に一致するので，ユニフィケーションが成功し，Prologはyesを返す．この例は極めて単純だが，もう一捻りしてみよう．ユニフィケーションはルールの右側だけでなく左側でも起こる．次の例を見てほしい．

```
| ?- dorothy(One, Two, Three).

One = lion
Three = bear
Two = tiger

yes
```

この例では，推論が一段階増えている．dorothyの定義の右辺のゴール部では，PrologはX，Y，Zをそれぞれlion，tiger，bearにユニファイする．またヘッドである左辺では，X，Y，Zをそれぞれ One，Two，Threeにユニファイして結果を出力している．

では，最後のルールtwin_cats/2に移ろう．このルールは，「XとYの両方が猫であることを証明できれば，twin_cats(X, Y)は真である」と言っている．試してみよう．

```
| ?- twin_cats(One, Two).

One = lion
Two = lion ?
```

Prologは最初の例を返している．lionとlionはどちらも猫である．どのようにしてこの答えに到達したのかを見てみよう．

1. 質問はtwin_cats(One, Two)だった．Prologは，OneをXに，TwoをYにバインドする．これを解くため，Prologは各ゴールをそれぞれ調べていく必要がある．
2. 最初のゴールはcat(X)（Xは猫である）だ．
3. これにマッチする事実は2つある．cat(lion)とcat(tiger)だ．Prologは最初の事実を試すため，Xをlionにバインドして，次のゴールに移る．
4. Prologはcat(Y)のYを適切な値へバインドしようとする．このゴールは，lionを選択することによって，最初のゴールとまったく同じ要領で解くことができる．
5. 両方のゴールが満たされたので，ルールは成功する．Prologは，ルールを成功させたOneとTwoの値を表示し，yesを返す．

さて，ルールを真にする最初の答えが見つかった．1つ答えが見つかれば十分な場合もあるが，2つ以上の答えが必要になる場合もある．すべての答えに一つひとつ手順を追ってたどり着くには；を，残りの答えをすべて一度に得るにはaを押す．

```
Two = lion ? a

One = lion
Two = tiger

One = tiger
Two = lion
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
One = tiger
Two = tiger

(1 ms) yes
```

Prolog は，ゴールと対応する事実から得られた情報を用いて，X と Y のすべての組み合わせを網羅している点に注意してほしい．後で見るように，ユニフィケーションを使うとデータの構造に基づいて高度なマッチングを行うこともできる．1 日目はこのくらいにしておく．2 日目ではもう少し骨のある問題に取り組んでみる．

Prolog の実用例

Prolog のような形で提示された「プログラムなるもの」を見ると，少し面食らうに違いない．Prolog では，詳細な手順を追ったレシピが存在することは少なく，完成時にフライパンから取り出すケーキに関する記述があるだけだ．筆者が Prolog を学習するにあたっては，Prolog を実際に使っている人にインタビューしたことが大変役に立った．研究プロジェクトでイルカを使った実験のスケジュールの作成に Prolog を使用した Brian Tarbox 氏に話を聞いた．

イルカ研究者 Brian Tarbox 氏のインタビュー

Bruce　Prolog の学習経験についてお聞かせください．

Brian　私が Prolog を学んだのは，1980 年代の後半，ハワイ大学マノア校の大学院に在籍していたときでした．当時私は，Kewalo Basin 海洋哺乳類研究所でバンドウイルカの認識能力について研究していました．その頃，研究所で行われていた議論の大半は，イルカの思考方法についての理論に関するものでした．我々は主に，Akeakamai（略称 Ake）という名前のイルカを研究対象としていました．大半の議論が，次のような具合に始まりました．「そうだね．Ake はこういう状況をおそらく認識しているんじゃないのかな」私は，修士論文に，研究者が信じている Ake の現実世界の認識に合致する実行可能なモデルを作成してみようと決めていました．我々が進めていた研究のごく一部でもモデル化したいと思っていました．このような実行可能なモデルによって Ake の実際の行動を予測できれば，Ake の思考方法に関する我々の理論にある程度の自信を持つことができます．

Prolog は素晴らしい言語ですが，流儀に十分慣れないうちは，返す結果がなんだか奇妙に思えることがあるでしょう．私は，Prolog に最初に触れたときの経験をよく思い出すことがあります．コードを書いていて `x = x + 1` という行を入力したときでした．Prolog が「no」と応答してきたのです．プログラミング言語が「no」とだけ返すことは普通ありません．間違った答えを返したり，コンパイルエラーになることはありますが，反論してくる言語など初めてでした．そこで，Prolog のサポートに連絡して，「変数の値を変更しようとすると『no』と言ってくるのですが」と質問してみたのです．すると，「どうして変数の値を変更する必要があるのですか」と逆に聞かれたのです．「変数の値を変更することを許さない言語なんてあるのか！」という感じでした．Prolog を心底理解すると，

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

変数は特定の値を持つか未バインド状態のどちらかだと納得できるのですが，当時はそうした考えはしっくりきませんでした．

Bruce　Prologをどのように使ったのですか？

Brian　私は2つのメインシステムを開発しました．イルカのシミュレータと実験スケジューラです．研究所では4頭のイルカに対して一日4つの実験を行っていました．研究用のイルカは驚くほど数が少ないのです．それぞれのイルカが異なる実験に使われており，実験ごとに異なるスタッフが必要でした．イルカの調教ができる人は，数人しかいませんでした．データの記録係ができる人はかなりいましたが，トレーニングが必要でした．大半の実験は6〜12人のスタッフが必要でした．院生，学部の学生，Earthwatch（環境NPO）のボランティアなどが作業にあたりました．すべてのスタッフがそれぞれのスケジュールを抱えており，持つスキルもそれぞれ違っていました．すべてのスタッフをうまく活用して，すべての作業を確実に終わらせるスケジュールを立てるには，1人のスケジュール作成専任スタッフが必要になっていました．
そこで，私はPrologでスケジュール作成プログラムを構築してみることにしました．やってみると，これがまさにPrologのために作られたような問題でした．各スタッフのスキル，スケジュール，各実験の要件を記述する事実を定義しました．あとは基本的にPrologに「スケジュールを作成せよ」と命令するだけでした．Prologは，実験に必要な各作業について，必要なスキルを備えた空いているスタッフを見つけて，そのスタッフを当該作業に割り当てて（バインドして）くれました．実験の要件が満たされるか，要件を満たすことができないと分かるまで，上記の処理が繰り返されます．妥当なバインドが見つからないときは，直前にバインドした内容を解除して，別の組み合わせを試してくれました．最終的にPrologは，妥当なスケジュールを見つけ出すか，実験の制約が大きすぎることを教えてくれました．

Bruce　イルカに関連する事実，ルール，あるいは表明のなかで，本書の読者にとって役に立ちそうな面白い例が何かありますか？

Brian　イルカのシミュレータがAkeの実際の行動を示唆してくれたことがありました．Akeは身振り手振りの言葉に反応します．例えば，「輪をくぐれ」とか「尾ひれでボールに触れろ」などです．Akeに指示を与えると，彼女がそれに答えるという具合です．
私の研究の1つは，“not”（〜ではない）などの簡単な単語を教えてみることでした．これからお話しするケースでは，“touch not ball”は，「ボール以外のものにさわれ」という意味でした．これはAkeにとっては難しい問題でしたが，研究はしばらく順調に進んでいました．ところがあるとき，この指示を与えるといつもAkeが単に水中に潜るだけという行動をとるようになったのです．我々にはその意味がまったく分かりませんでした．イルカにその行動の理由を尋ねるわけにはいかないので，こうした状況になると本当にいらいらします．そこで，Akeにやらせたのと同じ訓練をイルカシミュレータに実行させてみたところ，面白い結果が得られたのです．イルカは大変利口ですが，一般に，最も簡単な答えを見つけようとします．イルカシミュレータに同じヒューリスティックを与えたところ，Akeのジェスチャー言語に水槽内の窓の1つに対応するある単語が含まれてい

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ることが分かりました．その単語は滅多に使われないので，ほとんどの訓練師が忘れていたのです．イルカシミュレータが “window”（窓）は “not ball”（ボール以外）に対する正しい答えであることを見つけてくれたのです．それは，“not hoop”（輪をくぐるな），“not pipe”（鳴くな），および “not frisbee”（フリスビー以外）に対する正しい答えでもありました．他のモノについては，こうしたケースが起こらないように，訓練のたびに水槽の中のモノを入れ替えていたのですが，当然窓を外すわけにはいきませんでした．実は，Ake は水槽の底に潜ってその窓の横にいたのです．その窓はこちらには見えませんでしたが．

Bruce　Prolog のどのような点が最も気に入っていますか？

Brian　宣言型プログラミングモデルは非常に魅力的です．一般に，問題を記述することができれば，問題を解いたことになるからです．ほとんどの言語では，「言っていることは分かるだろ．そのとおりにやればいいんだよ！」とコンピュータにかみついてしまうことがあります．C や C++ の “semicolon expected”（セミコロンが抜けている）といった類のコンパイルエラーなどはまさに象徴的で，本当にいらいらさせられます．「セミコロンが抜けていると分かっているなら，挿入してみてエラーがなくなるかどうか試してみたらどうなんだ」と思ってしまいます．Prolog では，スケジュールを作成したければ，「こういう感じのスケジュールを組みたいんだけど，作ってくれるかな」と言うだけで，あとはすべてやってくれます．

Bruce　逆に，Prolog で最も大きな問題点は何ですか？

Brian　Prolog の問題に対するアプローチは「全か無か」方式のように思えます．少なくとも，私が取り組んでいた問題に対してはそうでした．実験スケジュールの作成問題では，30 分くらいあれこれ考えた末に，素晴らしいスケジュールを提示してくれるか，単に「no」を返すかです．「no」が返された場合は，その日のスケジュールは制約が多すぎて完全な答えが見つからなかったことを意味します．しかし，部分的な答えを教えてくれたり，制約が大きかった部分に関する情報を与えてくれることはありません．

この事例は Prolog の非常に強力な構想をよく示している．Prolog では問題の解法を記述する必要はない．問題を記述するだけでよい．そして，問題を記述するための言語は純粋な論理だけだ．事実と推論から始めれば，あとは Prolog がやってくれる．Prolog のプログラムは高レベルの抽象化を実現する．この事例のスケジュール作成と行動パターンは，Prolog が得意とする問題だ．

1 日目で学んだこと

1 日目では，Prolog の基本構成要素について学んだ．Prolog では，答えに至る道筋を手順を追ってコーディングするのではなく，純粋な論理を使って知識をコーディングする．あとは Prolog がその知識を組み合わせて答えを見つけてくれる．我々プログラマは，知識ベースに論理を組み込んで，それに対して質問をするという形になる．

初日の今日は，いくつか知識ベースを構築したあと，コンパイルして質問してみた．質問には 2 つの形式があった．ひとつは，質問に指定された事実に対して，Prolog が真が

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

偽かを答えるもの．もうひとつは，1 つ以上の変数を含む質問に対して，Prolog がそれらの事実を真にするようなすべての可能性を答えるというものだった．

Prolog はルールの節を順番に参照してルールをこなしていく．任意の節に対して，あらゆる変数の組み合わせを試して，各ゴールが満足されるようにする．Prolog のプログラムはすべてこのように動作する．

2 日目以降では，さらに複雑な推論を行う．数値計算，および複雑なデータ構造体（リスト）の使い方や，リストを繰り返し処理する方法についても学ぶ．

セルフスタディ 1 日目

探してみよう：

- 無償の Prolog チュートリアル
- サポートフォーラム（複数ある）
- Prolog のオンラインリファレンス（自分が使っている処理系に合ったものを選ぶこと）

試してみよう：

- 自分のお気に入りの本と作家を表す簡単な知識ベースを作成せよ．
- 作成した知識ベースから特定の作家の著作をすべて検索せよ．
- 音楽家と楽器を表す知識ベースを作成せよ．また，音楽家と音楽のジャンルを表す知識ベースも作成せよ．
- ギターを演奏するすべての音楽家を検索せよ．

4.3　2 日目：Wapner 判事まで 15 分

『レインマン』の主人公レイモンドは，人気テレビ番組 “The People’s Court” の気むずかしい Wapner 判事に執着している．大半の自閉症患者がそうであるように，レイモンドも見慣れたものすべてに執着する．で，“The People’s Court” の Wapner 判事が気に入ってしまった．皆さんは，ここまで Prolog という不可解な言語の勉強を進めてきて，そろそろいろいろなことが分かり出す頃かもしれない．ことによると，あなたは何でも瞬時に理解してしまう天才肌の人かもしれないが，そうでなくても大丈夫だ．Wapner 判事の登場までまだ 15 分はある．あせらずに行こう．今日は道具箱にもう少し道具が必要になる．再帰，数値計算，リストの使い方について学ぶのだ．それでは始めよう．

再帰

Ruby と Io は命令型プログラミング言語だ．したがってアルゴリズムの各ステップを逐一記述する必要がある．Prolog は本書に登場する最初の宣言型言語だ．リストやツリーなどのコレクションを扱う際には，繰り返しではなく再帰を使うことが多い．ここでは再帰について説明し，再帰を用いて基本的な推論を伴う問題をいくつか解き，同じ手法をリストと数値計算にも適用してみる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

次のデータベースを見てほしい．Waltons（ウォルトン家．1963 年の映画および以降のテレビシリーズ番組）の家系図を表したものだ．父子関係を表現しており，そこから祖先関係を推論している．祖先は父，祖父，曾祖父を意味するため，ルールをネストさせるか繰り返しを使う必要がある．Prolog は宣言型言語なのでネストを使う．ancestor 節内の 1 つの節で再度 ancestor を使用する．この例では ancestor(Z, Y) が再帰的サブゴールになっている．以下に知識ベースを示す．

▶ code/prolog/family.pl

```
father(zeb,          john_boy_sr).
father(john_boy_sr, john_boy_jr).

ancestor(X, Y) :-
    father(X, Y).
ancestor(X, Y) :-
    father(X, Z), ancestor(Z, Y).
```

father は，再帰的サブゴールを使えるようにするための，元となる事実関係だ．ルール ancestor/2 には 2 つの節がある．1 つのルールが複数の節で構成されている場合，いずれかの節が真であれば，そのルールは真となる．サブゴールの間のカンマは and 条件，各節の間のピリオドは or 条件と考えればよい．最初の節は「X が Y の father であれば，X は Y の ancestor である」と言っている．これは単純明快な関係だ．このルールは次のようにして試すことができる．

```
| ?- ancestor(john_boy_sr, john_boy_jr).

true ?

no
```

Prolog は true を返す．john_boy_sr は，john_boy_jr の ancestor だ．この最初の節は事実に依存している．

2 番目の節 ancestor(X, Y) :- father(X, Z), ancestor(Z, Y). はもう少し複雑だ．この節は，「X が Z の father であり，なおかつ，その Z が Y の ancestor であることを証明できれば，X は Y の ancestor である」と言っている．

ふう．では 2 番目の節を使ってみよう．

```
| ?- ancestor(zeb, john_boy_jr).

true ?
```

そのとおり．zeb は john_boy_jr の祖先だ．いつもどおり，質問に変数を含めることもできる．

```
| ?- ancestor(zeb, Who).

Who = john_boy_sr ? a

Who = john_boy_jr

no
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:41:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

この答えから，zebはjohn_boy_jrとjohn_boy_srの祖先であることが分かる．ancestor述語は，次のように逆向きにも動作する．

```
| ?- ancestor(Who, john_boy_jr).

Who = john_boy_sr ? a

Who = zeb

(1 ms) no
```

これは素晴らしい．知識ベースのこのルールを2つの目的，すなわち祖先を見つける場合と子孫を見つける場合に使えるからだ．

ここで1つ注意してほしいことがある．再帰的サブゴールを使うときは，慎重に行う必要がある．というのは，再帰的サブゴールを使うたびにスタック領域が消費されるため，いつかはメモリ不足になる可能性があるからだ．宣言型言語はこの問題を末尾再帰最適化と呼ばれる手法で解決していることが多い．再帰的サブゴールを再帰ルールの末尾に置くことができれば，Prologはコールスタックを破棄するよう呼び出しを最適化できるので，メモリの使用量を一定に保つことができる．上記の例は，再帰的サブゴールancestor(Z, Y)が再帰的ルールの最後のゴールになっているので，末尾再帰である．Prologプログラムがスタック領域不足でクラッシュするようになったら，末尾再帰を使って最適化する方法を検討する時期であるということだ．

最後に注意点もクリアになったところで，リストとタプルについて見ていくことにしよう．

リストとタプル

リストとタプルはPrologのプログラミングで極めて重要な部分だ．リストは[1, 2, 3]，タプルは(1, 2, 3)のように指定する．リストは可変長の入れ物であり，タプルは固定長の入れ物だ．リストもタプルも，ユニフィケーションの観点から考えると，極めて強力な道具となる．

ユニフィケーションその2

Prologは変数をユニファイするとき，左辺と右辺を一致させようと試みる．2つのタプルは，要素の数が同じで各要素が一致するときにマッチする．いくつか例を見てみよう．

```
| ?- (1, 2, 3) = (1, 2, 3).

yes
| ?- (1, 2, 3) = (1, 2, 3, 4).

no
| ?- (1, 2, 3) = (3, 2, 1).

no
```

すべての要素がユニファイすれば，2つのタプルはユニファイする．最初のタプルは完全に一致する．2番目のタプルは要素の数が異なる．3番目のタプルは同じ要素が同じ順

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

序で並んでいない．変数を入れてみよう．

```
| ?- (A, B, C) = (1, 2, 3).

A = 1
B = 2
C = 3

yes
| ?- (1, 2, 3) = (A, B, C).

A = 1
B = 2
C = 3

yes
| ?- (A, 2, C) = (1, B, 3).

A = 1
B = 2
C = 3

yes
```

変数はどちら側にあっても関係ない．Prolog が両辺を一致させることができれば，両者はユニファイする．いくつか質問してみよう．タプルの場合と同じように動作する．

```
| ?- [1, 2, 3] = [1, 2, 3].

yes
| ?- [1, 2, 3] = [X, Y, Z].

X = 1
Y = 2
Z = 3

yes
| ?- [2, 2, 3] = [X, X, Z].

X = 2
Z = 3

yes
| ?- [1, 2, 3] = [X, X, Z].

no
| ?- [] = [].

yes
```

3 番目と 4 番目の例は面白い．`[X, X, Z]` と `[2, 2, 3]` はユニファイする．`X = 2` とすることで，両者を一致させることができるからだ．一方，`[1, 2, 3] = [X, X, Z]` はユニファイしない．1 番目と 2 番目の両方が `X` になっているが，1 番目の値と 2 番目の値は異なるからだ．リストには，タプルにはない機能がある．`[Head|Tail]` という構文でリストを分割できるのだ．この構文でリストをユニファイすると，`Head` がリストの先頭

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

要素に，Tail が残りのすべての要素にバインドされる．次に例を示す．

```
| ?- [a, b, c] = [Head|Tail].

Head = a
Tail = [b,c]

yes
```

[Head|Tail] は空リストとはユニファイしないが，1 要素のリストとはユニファイする．

```
| ?- [] = [Head|Tail].

no
| ?- [a] = [Head|Tail].

Head = a
Tail = []

yes
```

さまざまな組み合わせを使えば複雑にすることもできる．

```
| ?- [a, b, c] = [a|Tail].

Tail = [b,c]

(1 ms) yes
```

この例では a にマッチしたあと，残りすべてと Tail とがユニファイする．このテール部分をさらにヘッドとテールに分割することもできる．

```
| ?- [a, b, c] = [a|[Head|Tail]].

Head = b
Tail = [c]

yes
```

3 番目の要素を取り出すには次のようにする．

```
| ?- [a, b, c, d, e] = [_, _|[Head|_]].

Head = c

yes
```

_はワイルドカードで，何とでもユニファイする．基本的には「この位置に何がきても構わない」という意味だ．上の例では，先頭の 2 つの要素をスキップし，残りをヘッドとテールに分割するよう指示している．結果として，Head に 3 番目の要素が取り出され，末尾の_にテールが取り出されるが，その部分は無視される．

最初はこのくらい知っておけば十分だろう．ユニフィケーションは強力な道具だ．ユニフィケーションにリストとタプルを組み合わせることによって，さらに強力な道具と

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

なる．

これで，Prolog の中心となるデータ構造について基本的な理解は得られたと思う．次に，これらの要素をルールおよび表明と組み合わせて，基本的な数値計算を論理式で表現してみよう．

リストと数値計算

次の例では，再帰と数値計算を使ってリストを処理する例をお見せしよう．具体的には，要素の数え上げ，合計，平均の計算を実行する．次の 5 つのルールですべての処理が実行される．

▶ code/prolog/list_math.pl

```
count(0, []).
count(Count, [Head|Tail]) :- count(TailCount, Tail), Count is TailCount + 1.

sum(0, []).
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.

average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

最も簡単な例は count だ．次のように使う．

```
| ?- count(What, [1]).

What = 1 ? ;

no
```

要素を数え上げるルールは極めてシンプルだ．まず，空リストの要素数は 0 である．次に，リストの要素数はテール部の要素数に 1 を加えたものだ．この 2 つのルールが機能する様子を手順を追って見てみよう．

- count(What, [1]) という質問をする．この質問は，リストが空ではないので，最初のルールとはユニファイしない．そこで，2 番目のルールのゴール count(Count, [Head|Tail]) を満足させることを考える．質問とこのゴールはユニファイして，What が Count に，Head が 1 に，Tail が [] にバインドされる．
- ユニフィケーション発生後の最初のゴールは count(TailCount, []) だ．このサブゴールの証明を試みる．今度は最初のルールとユニファイする．これにより TailCount が 0 にバインドされる．これで最初のゴールは満たされたので，2 番目のゴールに移る．
- 今度は Count is TailCount + 1 を評価する．この評価により各変数をユニファイできる．すなわち，TailCount が 0 にバインドされるので，Count が 0 + 1，つまり 1 にバインドされる．

以上だ．再帰の過程は具体的には定義していない．論理ルールを定義しただけだ．次に，リストの各要素の合計を求めてみる．以下にルールを再掲する．

```
sum(0, []).
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
sum(Total, [Head|Tail]) :- sum(Sum, Tail), Total is Head + Sum.
```

このコードは要素を数え上げるルールとまったく同じように動作する．基礎になるケースと再帰的ケースの 2 つの節があるところも同じだ．使い方も似ている．

```
| ?- sum(What, [1, 2, 3]).

What = 6 ? ;

no
```

このコードを命令型として見ると，sum は再帰言語に期待されるとおりの動作をする．空リストの sum はゼロであり，空リスト以外のリストの合計は，Head に Tail の sum を加算した値となる．

しかし別の解釈もある．つまり，Prolog にリストの各要素の合計を計算する方法を指示したのではなく，単に合計をルールとゴールとして記述しただけと考えるのである．ゴールを満たすには，論理エンジンがいくつかのサブゴールを満たす必要がある．宣言的な解釈は次のようになる．「空リストの合計はゼロである．テールの合計とヘッドを足したものが Total であることを証明できれば，リストの合計は Total である」．ゴールとサブゴールを証明することで再帰の代わりにしている．

同様に要素の数え上げの解釈は次のようになる．「空リストの要素数はゼロである．リストの count は，Head の要素数に Tail の count を足したものである」．

論理関係と同じように，これらのルールも組み合わせることができる．例えば sum と count を組み合わせると平均を計算できる．

```
average(Average, List) :- sum(Sum, List), count(Count, List), Average is Sum/Count.
```

すなわち，次のことが証明できれば，List の average は Average である．

- List の sum は Sum である．
- List の count は Count である．
- Average は Sum/Count である．

実行してみると期待どおりに動く．

```
| ?- average(What, [1, 2, 3]).

What = 2.0 ? ;

no
```

ルールを両方向に使う

ここまでの説明で，再帰の動作原理についてかなり理解できたはずだ．ここで，少し視点を変えて，append という単純明快なルールについて見ていきたい．ルール append(List1, List2, List3) は，List3 が List1 + List2 であるとき真となる．さまざまな使い方のある強力なルールだ．

このコードは短いがすごい威力を秘めており，実にさまざまな方法で活用できる．次

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

のように「嘘発見器」にもなる．

```
| ?- append([oil], [water], [oil, water]).

yes
| ?- append([oil], [water], [oil, slick]).

no
```

リストを作成することもできる．

```
| ?- append([tiny], [bubbles], What).

What = [tiny,bubbles]

yes
```

リストの差分をとることもできる．

```
| ?- append([dessert_topping], Who, [dessert_topping, floor_wax]).

Who = [floor_wax]

yes
```

リストを分割する方法を洗い出すこともできる．

```
| ?- append(One, Two, [apples, oranges, bananas]).

One = []
Two = [apples,oranges,bananas] ? a

One = [apples]
Two = [oranges,bananas]

One = [apples,oranges]
Two = [bananas]

One = [apples,oranges,bananas]
Two = []

(1 ms) no
```

1 つのルールで 4 つのことができたわけだ．このようなルールを作るにはかなりのコードが必要そうに思えるかもしれない．実際，どのくらいのコードになるのか確かめてみよう．Prolog の append を concatenate という名前で書き直してみる．次のような手順を踏む．

1. 空リストを List1 に連結するルール concatenate(List1, List2, List3) を書く．
2. List1 の 1 つの要素を List2 に連結するルールを追加する．
3. List1 から要素を 2 つとって List2 に連結するルールを追加する．同様に要素を 3 つとって List2 に連結するルールも追加する．
4. ルールの一般化を試みる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

では始めよう．手順 1 では空リストを List1 に連結する．これは非常に簡単なルールだ．

▶ code/prolog/concat_step_1.pl

```
concatenate([], List, List).
```

問題ない．最初の引数が空リストで次の 2 つの引数が同じであれば，concatenate は真である．

次のとおり，うまく動作する．

```
| ?- concatenate([], [harry], What).

What = [harry]

yes
```

手順 2 に進もう．List1 の先頭要素を List2 の頭に連結するルールを追加する．

▶ code/prolog/concat_step_2.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
```

concatenate(List1, List2, List3) で，List1 をヘッドとテールに分割し，テールは空リストにする．List3 をヘッドとテールに分割し，List1 をヘッドに，List2 をテールにする．知識ベースをコンパイルするのを忘れないように．次のとおり，うまく動作する．

```
| ?- concatenate([malfoy], [potter], What).

What = [malfoy,potter]

yes
```

ここまでくると，長さ 2，および長さ 3 のリストを concatenate するルールも同じ要領で定義できる．これらのルールも同じように動作する．

▶ code/prolog/concat_step_3.pl

```
concatenate([], List, List).
concatenate([Head|[]], List, [Head|List]).
concatenate([Head1|[Head2|[]]], List, [Head1, Head2|List]).
concatenate([Head1|[Head2|[Head3|[]]]], List, [Head1, Head2, Head3|List]).
```

```
| ?- concatenate([malfoy, granger], [potter], What).

What = [malfoy,granger,potter]

yes
```

ここまでで，基本ケース，および各サブゴールが最初のリストを分割して短縮し，3 番目のリストを長くする戦略が用意できた．2 番目のリストはそのままで変わらない．これで，結果を一般化する情報が十分に得られた．以下にネストされたルールを使った

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

concatenate を示す．

```
concatenate([], List, List).
concatenate([Head|Tail1], List2, [Head|Tail3]) :-
  concatenate(Tail1, List2, Tail3).
```

この簡潔で小さなコードブロックは，信じられないくらい簡単に説明できる．最初の節は「空リストに List を連結すると List になる」という意味だ．2 番目の節は「List1 と List3 のヘッドが同じで，なおかつ List1 のテールに List2 を連結したものが List3 のテールになることを証明できれば，List1 に List2 を連結すると List3 になる」という意味だ．このシンプルでエレガントな解法は，Prolog が強力であることの証である．

では，concatenate([1, 2], [3], What) という質問がどのように処理されるか見てみよう．各手順でどのようにユニフィケーションが行われるのかを見ていく．ルールをネストしているため，サブゴールの証明を試みるたびに変数のコピーが新たに作られることを覚えておいてほしい．重要な変数には末尾に文字を付けて別の変数であることが分かるようにした．各手順では，Prolog が次のサブゴールを証明しようとするとき何が起こるのかを示す．

- 次の質問から開始する．
  `concatenate([1, 2], [3], What)`
- 最初のルールは適用されない．[1, 2] は空リストではないからだ．
  `concatenate([1|[2]], [3], [1|Tail2-A]) :- concatenate([2], [3], [Tail2-A])`
  2 番目のテール（Tail2-A）を除き，すべてがユニファイする．次のゴールに移る．右側をユニファイしてみよう．
- concatenate([2], [3], [Tail2-A]) にさらにルールを適用してみよう．次のようになる．
  `concatenate([2|[]], [3], [2|Tail2-B]) :- concatenate([], [3], Tail2-B)`
  Tail2-B は Tail2-A のテールである点に注意してほしい．元の Tail2 とは異なる．ここで再度，右側をユニファイする．
- `concatenate([], [3], Tail2-C) :- concatenate([], [3], [3]) .`
- これで，Tail2-C が [3] であることが分かった．あとはこれまでの処理経路を逆向きに辿っていけばよい．各手順で Tail2 に値を入れながら，3 番目のパラメータを見ていく．Tail2-C は [3] なので，[2|Tail2-2] は [2, 3] となり，最後に [1|Tail2] は [1, 2, 3] となる．したがって What は [1, 2, 3] である．

この例で，Prolog はたくさんの仕事をしてくれている．理解できるまで上の手順を読み返しておいてほしい．ネストされたサブゴールをユニファイするのは，本書に出てくる高度な問題を解くために最も重要な概念である．

以上，Prolog の中で最も機能が充実している部分について詳しく解説した．少し時間をとって，紹介した解法を実際に試して，理解を確実にしておいてほしい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

2 日目で学んだこと

2 日目は，Prolog がデータを整理するために使う基本構築ブロックであるリストとタプルについて解説した．ルールをネストさせて，他の言語ならループで処理するような問題を表現（記述）できるようにした．また，Prolog のユニフィケーション，および Prolog が :- または=の左右両辺をマッチングする方法についても深く掘り下げて説明した．Prolog でルールを書くとき，実はアルゴリズムの代わりに論理ルールを記述しているのであり，その論理ルールに従って Prolog に解を求めさせているのだということを理解した．

数値計算も行った．基本的な演算とネストされたサブゴールを用いて合計と平均を計算した．

最後にリストの使い方について学んだ．リスト内の複数の変数を変数とマッチさせた．とりわけ重要なのは，`[Head|Tail]` というパターンを用いて，リストのヘッドと残りの要素を変数にマッチさせる方法だ．この手法を用いてリストの再帰処理を行った．これらの構築ブロックは，3 日目で複雑な問題を解くための基盤となる．

セルフスタディ 2 日目

探してみよう：

- フィボナッチ数列と階乗計算の実装．動作方法を説明せよ．
- Prolog の実世界での使用例．現在，どのような問題を解くのに Prolog が使われているか？

少し高度な問題に取り組んでみたいなら，次の問題に挑戦してみるとよい．

- ハノイの塔を実装し，その動作方法を説明せよ．
- 「not」（〜でない）という表現を扱うときの問題点を指摘せよ．Prolog では否定表現に気をつけなければならないが，それはなぜか．

試してみよう：

- リストの各要素を逆順に並べよ．
- リスト内で最も小さな要素を見つけよ．
- リスト内の要素をソートせよ．

4.4　3 日目：ラスベガスをぶっとばせ

ここまで読み進めてきた皆さんには，筆者が Prolog をレインマン（特殊な才能を持つ自閉症患者）に喩えた理由が分かってきたはずだ．ときどき理解しがたいことはあるが，プログラミングをこんな風に考えるのはすごいことだ．『レインマン』に，レイがカードカウンティング[*4]ができることを弟が知るシーンがある．筆者のお気に入りのシーンだ．レイと弟はラスベガスに出かけて行き，大勝ちしてほとんど胴元をつぶしそうになる．3

[*4] ［訳注］カードゲームのテクニックのひとつ．配られたカードを記憶して，残りのカードを予測する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

日目は，思わず笑顔になるような Prolog の一面をお見せしよう．筆者はこの章で紹介する例をコーディングしていて気が狂いそうだったが，同時にウキウキした爽快な気分にもなった．この章では，2 つの有名なパズルを解いてみる．制約の中で正しい手順を見つけ出すという Prolog お得意の分野だ．

これらのパズルを自力で解いてみたいという方もいるだろう．その場合は，各ゲームについて知っているルールを記述するようにしてみるとよい．手順を追った解法を Prolog に示す必要はない．まずは小さな数独から始め，最後の演習で，より大きなものを解いてもらう．次に，有名な 8 クイーン問題に挑戦する．

数独を解く

数独のコーディングは，筆者にとってはほとんど魔法のようだった．数独は，行，列，ボックスから成る格子状のマス目である．典型的な数独の問題では，9×9 の 81 マスの中に，埋められているマスと空いているマスがある．各マスには 1〜9 の数字が入る．数独を解くには，各行，列，ブロックに，1〜9 の数字が重複することなく 1 つずつ入るようにする．

ここでは 4×4 の数独から始めることにする．考え方はまったく同じだが，解は短くなる．では，やり方を説明しよう．4 行，4 列，4 ブロックのボードがあると考える．次の表に 4 つのブロックを示す．

1 1	1 1	2 2	2 2
1 1	1 1	2 2	2 2
3 3	3 3	4 4	4 4
3 3	3 3	4 4	4 4

最初に，質問がどのような形になるか決める必要がある．これは簡単だ．パズルとその答えを `sudoku(Puzzle, Solution)` という形式で表す．ユーザは，パズルをリストとして与える．その際，未知の数字（空いているマス）はアンダースコアで置き換える．

```
sudoku([_, _, 2, 3,
        _, _, _, _,
        _, _, _, _,
        3, 4, _, _],
        Solution).
```

解が存在していれば，Prolog はその解を示す．筆者はこのパズルを Ruby で解いてみたが，パズルを解くためのアルゴリズムを考える必要があった．Prolog では，その必要はない．ゲームのルールを与えればよいだけだ．以下にルールを示す．

- パズルが解けた場合，パズルを埋める数字と解の数字は一致していなければならない．
- 数独ボードはそれぞれが 1〜4 の値をとる 16 のマス目がある．
- ボードは，4 行，4 列，4 正方ブロックで構成される．
- 各行，各列，各ブロックのマス目が 1〜4 の数字で重複することなく埋められていれば，そのパズルは解けたことになる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:46:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

最初のルールから始めよう．パズルを埋める数字と解の数字は一致していなければならない．

▶ code/prolog/sudoku4_step_1.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle.
```

これでも実は一歩前進している．この「数独ソルバー」は，マス目がすべて埋まっているケースを解くことができるのである．

```
| ?- sudoku([4, 1, 2, 3,
             2, 3, 4, 1,
             1, 2, 3, 4,
             3, 4, 1, 2], Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

見栄えは良くないが，意図は十分に伝わる．16のマスを行単位で返している．でも，この制約だけではちょっと欲張りだ．

```
| ?- sudoku([1, 2, 3], Solution).

Solution = [1,2,3]

yes
```

この解は正解ではないが，ソルバーは有効な解答が存在すると言っている．明らかに，ボードは16個の要素に限定する必要がある．別の問題もある．マス目に入る値が制限されていないのだ．

```
| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

Solution = [1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6]

yes
```

解として妥当であるためには，数字は1〜4のいずれかでなければならない．この条件を入れておかないと，2つの問題を抱えることになる．第一に，妥当でない解を許してしまう可能性がある．第二に，各セルに入りうる数字を判定するための十分な情報がPrologに与えられていない．言い換えると，結果セットに根拠がないのだ．つまり，各セルに入りうる値を制限するルールを記述していないため，Prologは数字を推測することができない．

では，次のルールを加えることで，この2つの問題を解決してみよう．ルール2は，「ボードには16のマス目があり，各マス目には1〜4の数字が入る」と言っている．GNU Prologには，考えられる値を表現する組み込みの述語 `fd_domain(List, LowerBound, UpperBound)` が用意されている．この述語は，`List` 内のすべての値が `LowerBound` 以上 `UpperBound` 以下であれば真となる．ここでは，`Puzzle` 内の値がすべて1〜4であればよい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/prolog/sudoku4_step_2.pl

```
sudoku(Puzzle, Solution) :-
        Solution = Puzzle,
        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],
        fd_domain(Puzzle, 1, 4).
```

Puzzle を 16 変数のリストとユニファイし，マス目に入る数字を 1〜4 に制限した．これで，次のようにパズルが妥当でなければ no が返される．

```
| ?- sudoku([1, 2, 3], Solution).

no


| ?- sudoku([1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6], Solution).

no
```

さて，いよいよ解法の核心部分に入ってきた．ルール 3 は，「ボードは，行，列，正方ブロックで構成される」と言っている．そこで，パズルを，行，列，正方ブロックに分割する．ここで，マス目にこのような名前を付けた理由が分かる．行を記述するにはこの方法が最も簡単だからだ．

```
Row1 = [S11, S12, S13, S14],
Row2 = [S21, S22, S23, S24],
Row3 = [S31, S32, S33, S34],
Row4 = [S41, S42, S43, S44],
```

列も同様で，次のようになる．

```
Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],
```

正方ブロックも同様だ．

```
Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44].
```

ボードを分割できたので，次のルールに移ろう．すべての行，列，正方ブロックに重複する要素がなければ，そのボードは妥当となる．fd_all_different(List) は，List のすべての要素が異なっていれば真を返す．ここでは，すべての行，列，正方ボックスが妥当であることを判定するルールを作る必要がある．簡単なルールを使ってこれを実現してみる．

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
    valid(Tail).
```

この述語は，指定されたすべてのリストが異なっていれば妥当となる．最初の節では，空リストは妥当であると言っている．2番目の節では，最初の要素（リスト）の要素がすべて異なっており，残りのリストが妥当であれば，そのリストは妥当であると言っている．

あとは，valid(List) ルールを起動するだけである．

```
valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

信じられないかもしれないが，解法は以上だ．これで4×4の数独を解くことができる．

```
| ?- sudoku([_, _, 2, 3,
             _, _, _, _,
             _, _, _, _,
             3, 4, _, _],
             Solution).

Solution = [4,1,2,3,2,3,4,1,1,2,3,4,3,4,1,2]

yes
```

読みやすい形式にすると，次のようになる．

4 4	1 1	2 2	3 3
2 2	3 3	4 4	1 1
1 1	2 2	3 3	4 4
3 3	4 4	1 1	2 2

完成したプログラム全体を以下に示す．

▶ code/prolog/sudoku4.pl

```
valid([]).
valid([Head|Tail]) :-
    fd_all_different(Head),
    valid(Tail).


sudoku(Puzzle, Solution) :-
        Solution = Puzzle,

        Puzzle = [S11, S12, S13, S14,
                  S21, S22, S23, S24,
                  S31, S32, S33, S34,
                  S41, S42, S43, S44],

        fd_domain(Solution, 1, 4),

        Row1 = [S11, S12, S13, S14],
        Row2 = [S21, S22, S23, S24],
        Row3 = [S31, S32, S33, S34],
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Row4 = [S41, S42, S43, S44],

Col1 = [S11, S21, S31, S41],
Col2 = [S12, S22, S32, S42],
Col3 = [S13, S23, S33, S43],
Col4 = [S14, S24, S34, S44],

Square1 = [S11, S12, S21, S22],
Square2 = [S13, S14, S23, S24],
Square3 = [S31, S32, S41, S42],
Square4 = [S33, S34, S43, S44],

valid([Row1, Row2, Row3, Row4,
       Col1, Col2, Col3, Col4,
       Square1, Square2, Square3, Square4]).
```

Prolog らしさをまだ感じたことがない人は，この例が正しい方向に進むための良いきっかけになるはずだ．上の例で，プログラムはどこにあるのだろう．実は，プログラムは書かなかったのである．ゲームのルールを記述しただけである．つまり，「ボードには 16 のマス目があり，各マス目には 1〜4 の数字が入る」「1 つの行，列，または正方ボックス内で数字が重複することはない」というルールだ．このパズルは，30 行足らずのコードで解くことができた．数独を解くための戦略に関する知識は必要なかった．最後の演習で 9×9 の数独を解いてもらう．それほど難しくはないはずだ．

このパズルは Prolog が得意とする種類の問題の格好の例である．こうしたパズルでは，簡単に記述できるが解くのは難しい一連の制約が与えられる．もう 1 つ，著しく制約されたリソースを用いて解くパズル，8 クイーン問題を紹介しよう．

8 クイーン問題

8 クイーン問題とは，チェス盤上の 8 つのクイーンを，どのクイーンも縦，横，斜めの線を共有しない（お互いの利き筋に，どのクイーンもない）ように配置する問題だ．表面上は，つまらない問題のように思える．単なる子供の遊びだ．しかし視点を変えると，縦，横，斜めの線を制約されたリソースとみなすことができる．ソフトウェア業界には，こうした制約されたシステムを解く問題が山ほどある．では，この 8 クイーン問題を Prolog で解く方法を見ていこう．

まず，質問がどのような形になるか見てみる．各クイーンは，縦と横のタプル (Row, Col) として表現できる．Board は（クイーンの位置を示す）タプルのリストだ．eight_queens(Board) は，妥当な盤面が与えられると成功する．質問は次のような形式になる．

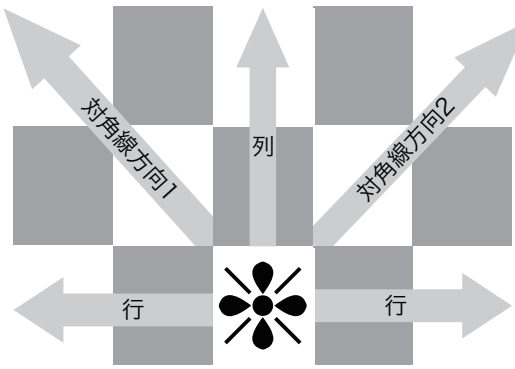
```
eight_queens([(1, 1), (3, 2), ...]).
```

このパズルを解くために満足する必要があるゴールを考えてみよう．解法を見ずにこのゲームをやってみたければ，これらのゴールを満たしてやればよい．完全な解法は章の最後にお見せする．

- 盤面には 8 つのクイーンがある．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 4.2　8 クイーン問題のルール

- 各クイーンの位置は，1〜8 行目，1〜8 列目である．
- どのクイーンも同じ行には存在しない．
- どのクイーンも同じ列には存在しない
- どのクイーンも（南西から北東の）同じ対角線上には存在しない．
- どのクイーンも（北西から南東の）同じ対角線上には存在しない．

行と列は一意でなければならないが，対角線にはもっと注意が必要だ．各クイーンは 2 つの対角線上に存在する．1 つは左下から右上に向かう対角線，もう 1 つは左上から右下に向かう対角線だ（図 4.2「8 クイーン問題のルール」参照）．これらのルールは比較的容易にコーディングできる．

前と同様，最初のルールから見ていこう．盤面には 8 つのクイーンが存在する．これはつまり，引数に指定するリストの長さが 8 でなければならないことを意味する．これは簡単にコーディングできる．前に出てきた `count` 述語，または Prolog 組み込みの述語 `length` を使えばよい．`length(List, N)` は，`List` が `N` 個の要素を持つ場合に成功する．今回は，各ゴールに対して実際に質問をして見せるのはやめて，問題全体を解決するために必要なすべてのゴールを順に解説していくことにする．最初のゴールは次のとおりだ．

```
eight_queens(List) :- length(List, 8).
```

次に，引数に指定したリストの各クイーンが妥当であることを確認する必要がある．そこで，クイーンが妥当かどうか判定するルールを作成する．

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).
```

`member` 述語は，ご想像どおりの動作をする．つまり，メンバーシップ（特定の集合に属するかどうか）をテストするのである．クイーンは，行と列がどちらも 1 以上 8 以下の整数であれば妥当となる．次に，盤面のクイーンがすべて妥当かどうかをチェックする．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).
```

空の盤面は妥当である．最初の項目が妥当なクイーンで，盤面の残りの項目が妥当であれば，その盤面も妥当となる．

次に進もう．次のルールは，2 つのクイーンが同じ行上に存在してはならない，というものだ．この 3 番目以降の制約を解決するには，ちょっとした手助けが必要になる．問題を記述しやすいようにプログラムを分解してみよう．行，列，対角線とは何だろうか．まずは行から．ここでは rows(Queens, Rows) という関数を作る．この関数は，各クイーンの Row 要素からなるリストと Rows とが等しい場合に真になるようにする．

```
rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).
```

このコードを理解するには少しばかり想像力が必要だ．空リストの rows は空リストだ．rows(Queens, Rows) は，Queens リストの最初のクイーンの Row が Rows の最初の要素と一致し，かつ Queens のテールの Row のリストが Rows のテールであれば，真になる．混乱するようであれば，いくつかテスト用のリストで試してみてほしい．列についてもまったく同じ関数を作ることができる．行の代わりに列を使えばよい．

```
cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).
```

ロジックは rows とまったく同じだ．違うのは，クイーン・タプルの 1 番目の要素ではなく 2 番目の要素をマッチさせる点だけだ．

次に進もう．次に対角線に番号を振る．対角線に番号を振る最も簡単な方法は，簡単な引き算と足し算を行う方法だ．北西の角を (1, 1) として，北西から南東に向かう対角線に Col-Row の値を割り当てる．これらの対角線を表す述語は次のようになる．

```
diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).
```

このルールは行および列と同じように動作するが，Diagonal is Col - Row という制約が追加されている．これはユニフィケーションではないので注意してほしい．これは，is という述語で，解に矛盾がないことを確認するものだ．同様に，北東から南西に向かう対角線を表す述語は次のようになる．

```
diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).
```

式は少し分かりづらいが，行と列の合計値が同じクイーンはすべて同じ対角線上に存在することに納得できるまで，いくつか値を試してみてほしい．以上で，行，列，対角線を記述するルールができた．あとは，すべての（クイーンについて）行，列，対角線が異

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

なっていることを確認すればよい．

まとめて見直せるように，解法全体を以下に示す．最後の 8 つの節で利き筋の確認をしている．

▶ code/prolog/queens.pl

```
valid_queen((Row, Col)) :-
    Range = [1,2,3,4,5,6,7,8],
    member(Row, Range), member(Col, Range).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

rows([], []).
rows([(Row, _)|QueensTail], [Row|RowsTail]) :-
  rows(QueensTail, RowsTail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  length(Board, 8),
  valid_board(Board),

  rows(Board, Rows),
  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),

  fd_all_different(Rows),
  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

この時点でプログラムを実行できるが，処理にかなり時間がかかる．組み合わせが多すぎて効率的に調べることができないからだ．しかし，少し考えてみると，各行に存在できるクイーンは 1 つだけであることが分かる．そこで，次のような盤面から始めることで，不要な処理を割愛できる．

```
| ?- eight_queens([(1, A), (2, B), (3, C), (4, D), (5, E), (6, F), (7, G), (8, H)]).

A = 1
B = 5
C = 8
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
D = 6
E = 3
F = 7
G = 2
H = 4 ?
```

これでも問題なく動作するが，依然としてプログラムの処理量はかなり多い．行の選択肢を減らすのは結構簡単だ．ついでに API も簡素化できる．以下に，若干最適化されたプログラムを示す．

▶ code/prolog/optimized_queens.pl

```
valid_queen((Row, Col)) :- member(Col, [1,2,3,4,5,6,7,8]).

valid_board([]).
valid_board([Head|Tail]) :- valid_queen(Head), valid_board(Tail).

cols([], []).
cols([(_, Col)|QueensTail], [Col|ColsTail]) :-
  cols(QueensTail, ColsTail).

diags1([], []).
diags1([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col - Row,
  diags1(QueensTail, DiagonalsTail).

diags2([], []).
diags2([(Row, Col)|QueensTail], [Diagonal|DiagonalsTail]) :-
  Diagonal is Col + Row,
  diags2(QueensTail, DiagonalsTail).

eight_queens(Board) :-
  Board = [(1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _)],
  valid_board(Board),

  cols(Board, Cols),
  diags1(Board, Diags1),
  diags2(Board, Diags2),

  fd_all_different(Cols),
  fd_all_different(Diags1),
  fd_all_different(Diags2).
```

冷静に見ると，大きな変更を 1 つ加えただけだ．Board を (1, _), (2, _), (3, _), (4, _), (5, _), (6, _), (7, _), (8, _) とマッチングさせて順列の数を大幅に削減し，行に関するすべてのルールも削除した．効果はある．筆者が長年使っている MacBook 上で実行した結果，すべての解が 3 分以内に得られた．

この例でも，最終結果は極めて満足のいくものだ．解法に関する知識はほとんど組み込んでいない．ゲームのルールを記述し，ちょっとしたロジックを適用して処理速度を若干向上させただけだ．Prolog 向きの適切な問題があれば，筆者は間違いなく Prolog の世界に入り込んでいくと思う．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

3 日目で学んだこと

3 日目の今日は，Prolog で用いた概念を組み合わせて，代表的なパズルを解いてみた．制約に基づく問題は，実世界にありがちな問題に見られる特徴の多くを備えている．制約条件を書き下して，それらを満たす解をガリガリ見つけていくというのがそれだ．SQL で 9 つの表を結合しようなどとは決して思わないだろうが，論理問題を解く場合は，その程度の制約はごく普通のことで驚くにもあたらない．

最初は数独のパズルだ．Prolog の解法は驚くほどシンプルだ．まず，16 の変数を行，列，正方ブロックに割り当てる．次に，各行，各列，各正方ブロック内で重複がないようにゲームのルールを記述する．すると Prolog があらゆる組み合わせを順序だてて調べ，あっという間に答えを導きだす．

次に，Prolog を用いて 8 クイーン問題を解いた．ここでもやはり，ゲームのルールをコーディングして，あとは Prolog に答えを計算させた．この 92 通りの答えを持つ定番のパズルは，計算量が極めて多いものだが，我々が考えた単純な方法でも数分以内に問題を解くことができた．

筆者は，高度な数独を解くために必要な技法やテクニックをすべて知っているわけではないが，Prolog を使えばその必要もない．ゲームのルールさえ知っていればよい．

セルフスタディ 3 日目

探してみよう：

- Prolog には，入出力機能もいくつか用意されている．変数を出力する述語 `print` を見つけよ．
- `print` を用いて正しい答えだけを出力する方法を考えよ．この述語はどのような方法で動作しているか？

試してみよう：

- 6×6（各ブロックの大きさは 3×2）および 9×9 のパズルも解けるように数独ソルバーを改良せよ．
- 解を見やすく出力するように数独ソルバーを改良せよ．

あなたがパズル好きなら，Prolog に夢中になるだろう．説明したパズルの解法を深く調べてみたいなら，手始めとして 8 クイーン問題がよいだろう．

- クイーンのリストを引数として受け取るようにして，8 クイーン問題を解け．ただし，各クイーンは，タプルではなく，1〜8 の整数で表し，クイーンの行はリスト内の位置として，クイーンの列はリスト内の値として取り出すようにせよ．

4.5 Prolog のまとめ

Prolog は，本書で紹介する言語のなかでも古くからある言語の 1 つだが，その概念は今でも興味深く意義のあるものだ．Prolog とは，論理によるプログラミングという意味だ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

本章では Prolog を用いてルールを処理した．ルールは節で構成され，節は一連のゴールで構成される．

Prolog プログラミングは，大きく 2 つの手順からなる．まず知識ベース（事実関係の論理表現と，問題領域における推論手順からなる）を構築し，次に知識ベースをコンパイルして，その問題領域に関する質問をする．一部の質問は表明であり，これに対して Prolog は `yes` または `no` で答える．一方，変数を含む質問もある．Prolog は，これらの質問が真になるように，変数に入る値の組み合わせを求める．

Prolog では，単純な代入ではなく，式の両辺の変数をマッチさせるユニフィケーションと呼ばれるプロセスを使用する．場合によっては，考えられる変数のあらゆる組み合わせを試して，変数をユニファイし，推論を進める．

長所

Prolog は，航空機のスケジューリングから金融派生商品まで，広範な問題に利用できる．Prolog は習得に時間がかかるが，要件の厳しい問題を解くことができるため存在価値は高い．

Brian Tarbox 氏のイルカの研究の話を思い起こしてほしい．彼は，現実世界に関する単純な推論から，イルカの行動に関する複雑な推論を飛躍的に前進させることができた．また，人的資源が非常に制約されているケースで，Prolog を使って，各スタッフの予定に合わせてスケジュールを作成することができた．こうした分野では，現在でも Prolog が実際に利用されている．

自然言語処理

Prolog はおそらく，言語認識処理に使われた最初の言語だ．具体的には，Prolog の言語モデルを使うことで，自然言語を入力として受け取り，事実と推論からなる知識ベースを適用して，複雑であいまいな言語をコンピュータでの処理に適した具体的なルールで表現することができる．

ゲーム

ゲームはどんどん複雑になっている．とりわけ，対戦相手や敵の動作をモデル化する処理は極めて複雑だ．Prolog モデルを使えば，システム内の他のキャラクタの動作を簡単に表現できる．また，敵の種類に応じて異なる動作を組み込むことで，よりリアルで楽しいゲームにできる．

セマンティック Web

セマンティック Web とは，Web 上のサービスや情報に意味を持たせることで，ユーザエージェントからのリクエストに簡単に答えられるようにする試みである．リソース記述言語（RDF）により，リソースの基本的な記述が可能だ．サーバはこれらのリソースをまとめて知識ベースを作成できる．この知識ベースと Prolog の自然言語処理機能を組み合わせることで，豊かなエンドユーザ体験を実現できる．この種の機能を Web サーバの分野で提供する多数の Prolog パッケージが存在する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:47:38 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

人工知能

人工知能（AI:Artificial intelligence）は，コンピュータに知能を与えることをテーマとしている．この知能はさまざまな形態をとるが，いずれの場合も，複雑なルールに基づいて「エージェント」が動作を修正する．Prologはとりわけ，ルールが具体的で形式論理に基づいている場合は，この分野で卓越した能力を発揮する．このためPrologは論理プログラミング言語と呼ばれることがある．

スケジューリング

Prologは，制約されたリソースを扱う作業のスケジューリングに優れた能力を発揮する．多くの人が，Prologを使って，オペレーティングシステムのスケジューラなど，高度なスケジューラを作成している．

短所

Prologは長年の使用に耐えて来た言語だ．とはいうものの，言語としていろいろな意味で時代遅れになっており，重大な制限もある．

実用性

Prologはその得意分野では優れた能力を発揮するが，それは論理プログラミングというかなり狭い分野である．つまりPrologは汎用言語ではない．言語設計に起因する制限もいくつかある．

大規模データセット

Prologは，一連のルールに対してマッチするすべての組み合わせを用いて，意志決定ツリーを深さ優先で探索する．さまざまな言語やコンパイラが，このプロセスを最適化するために，工夫をこらした素晴らしい仕事をしている．それでもこの戦略だと本質的に計算量が多くなる．特にデータセットが大規模になると，その傾向が強い．またそのため，プログラマは，データセットのサイズを管理可能な大きさに保つためにPrologが採用している仕組みを理解する必要がある．

命令型モデルと宣言型モデルの混在

関数型ファミリーに属する多くの言語，とりわけ再帰への依存度が大きい言語のプログラマはそうだが，Prologプログラマもやはり，再帰ルールの解決方法を理解する必要がある．ちょっと大きな問題を解くときでも末尾再帰ルールが欠かせないことがよくある．Prologでは，わずかなデータ量を超えるともうスケールしなくなるアプリケーションを比較的簡単に書いてしまうからだ．まともにスケールするようなルールをそつなく設計するには，Prologの動作原理を深く理解しておかねばならないことが多い．

終わりに

筆者は，本書でさまざまな言語を学んでいるとき，大型かなづちで毎回ネジを締めるような愚行をしていることにしょっちゅう気づいて，情けなくなったことがある．とりわけ，Prologは，なかなか理解が進まず苦労させられた言語だ．Prologに適している問題

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

が見つかったら，Prolog を活用してみよう．そうしたケースでは，ルールベースの言語である Prolog と汎用言語を組み合わせるのが一番だ．ちょうど，Ruby や Java に SQL を埋め込んで使うように．組み合わせ方に注意すれば，長期的には人より優位に立つことができるだろう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第5章

Scala

ぼくらは羊じゃないよ．
エドワード・シザーハンズ

ここまでで，3つの言語と3つの異なるプログラミングパラダイムを紹介した．Scalaは4つ目になるわけだ．Scalaはハイブリッドだ．つまり，意図的に，異なるプログラミングパラダイム間の橋渡しを試みた言語だ．具体的には，Javaなどのオブジェクト指向言語とHaskellなどの関数型言語の橋渡しを実現している．ある意味，Scalaはフランケンシュタインばりの怪物とも言えるが，実際には怪物ではない．「エドワード・シザーハンズ」を思い浮かべてほしい[*1]．

ティム・バートンが描いたこの幻想的な映画で，エドワードは，はさみの手を持つ，人間と機械のあいのことして登場する．以来，エドワードは筆者のお気に入りのキャラクタの一人である．エドワードはあの美しい映画の中で幻想的なキャラクタとして描かれていた．彼はぎこちない振る舞いをすることが多く，時として驚くような能力を見せるが，その表情はいつも独特だ．はさみの手のおかげで，信じられないことをやってのけたかと思うと，不器用で，落ち込んでいるときもある．新しくて異種なものは何でもそうだが，彼も誤解されることが多く，「本来あるべき正しい道から大きく外れている」と非難される．しかし，はにかみ屋の彼がいつになく強い語調で次のように言うシーンがある．「ぼくらは羊（臆病者）じゃないよ」．まったくそのとおりだ．

5.1 Scalaについて

コンピュータプログラムに求められる要件が複雑になるにつれ，言語も進化しなければならない．古いパラダイムは，ほぼ20年くらいの周期で，さまざまな概念を構成および表現するための新しい要求に対応しきれなくなる．そうなると新しいパラダイムの登場を待つしかないわけだが，ことはそう簡単ではない．

新しいプログラミングパラダイムが出現するたびに，いくつもの言語（1つではない）が登場する．最初に登場する言語はたいてい，際だって創造的だが，ひどく実用性に乏しい．オブジェクト指向の場合ならSmalltalk，関数型の場合ならLispがこれにあたる．次に，別のパラダイムの言語に，古いパラダイムの中で安全に作業しながら，新しい概念を吸収できる機能が組み込まれる．例えばAdaでは，カプセル化などのコアなオブジェクト指向の概念を手続き型言語の中に混在させることができる．そのうちに，古いパラダイムと新しいパラダイムの橋渡しをしながら，実用性もきちんと確保したハイブリッド言

[*1] 『シザーハンズ』．DVD．1990年，ティム・バートン監督作品．カリフォルニア州ビバリーヒルズ，20世紀フォックス．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

語が登場する．C++ がこれにあたる．次に，商用ベースの言語が登場する．Java や C#などがそうだ．最後に，新しいパラダイムを純粋な形で実装した成熟した言語が出現する．

Java との親和性

Scala は，橋渡し役はもちろん，それ以上の機能も備えている．Java との緊密な統合を実現しているため，Java プログラマは長年にわたる投資を無駄にせずに済む．例えば次のような利点がある．

- Java 仮想マシン上で動作するため，既存の環境で共存できる．
- Java のライブラリを直接使えるため，開発者は既存のフレームワークとレガシーコードを活用できる．
- Scala は Java と同様に静的に型付けされた言語なので，2 つの言語の基本的な考え方には共通するものがある．
- Scala のシンタックスは比較的 Java に近いので，開発者は基本的な構文をすぐに覚えることができる．
- オブジェクト指向と関数型の両方のプログラミングパラダイムをサポートしているため，プログラマは関数型のプログラミング概念を自分のコードに徐々に適用していくことができる．

盲目的に忠実というわけではない

自分の先祖にあたる言語の機能を取り入れている言語では，少しやりすぎて，先祖言語の弱点まで受け継いでしまっているものもある．一方 Scala の設計は，Java に大変よく似ているものの，Java から大きく乖離している部分があり，それはコミュニティにとっても良いことだろう．以下に示す主要な改良点は，Java からの乖離を図った部分である．

- 型推論．Java では，すべての変数，実引数，仮引数の型を宣言する必要がある．Scala は，できる限り変数の型を推論する．
- 関数型概念．Scala は，重要な関数型概念を Java に導入している．具体的には，既存の関数をさまざまな方法で用いて新しい関数を作ることができる．この章で説明する概念として，コードブロック，高階関数，高度なコレクションライブラリがある．Scala は基本的なシンタックスシュガーをはるかに超えた機能を実現している．
- 変更不能な変数．Java にも変更不能な変数用の修飾子はあるが滅多に使わない．Scala では，変数を変更可能にするかどうかを明示的に指定する必要がある．この指定は，並行コンテキストでのアプリケーションの動作方法に大きく影響する．
- 高度なプログラミング構文．Scala では基盤となる言語を用いて，役に立つ概念を追加している．この章では，並行性を実現するアクター，高階関数による Ruby 形式のコレクション，ファーストクラスの XML 処理について説明する．

説明に入る前に，Scala の開発経緯について紹介しておこう．Scala の開発者である Martin Odersky に，2 つのプログラミングパラダイムを結合すると決断するに至った経緯

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

を伺った．

Scala の開発者 Martin Odersky 氏へのインタビュー

Scala の開発者である Martin Odersky 氏は，École Polytechnique Fédérale de Lausanne（EPFL）（スイスに 2 校ある国立工科大学の 1 つ）の教授である．彼は，Java 総称性（ジェネリック）仕様の作業メンバーであり，`javac` リファレンスコンパイラの開発者だ．また，Scala 関連の良書『Scala スケーラブルプログラミング』[OSV08] の著者でもある．

Bruce　どうして Scala を開発したのですか?

Dr. Odersky　関数型言語とオブジェクト指向言語を統合すれば，非常に実用性の高い言語ができると確信していました．関数型プログラミングコミュニティでのオブジェクト指向プログラミングを見下したような態度にも，関数型プログラミングなど学者のお遊びで実用性がないとするオブジェクト指向プログラマの考えにも嫌気がさしていました．だから 2 つのパラダイムは統合できること，それによって新しく強力な何かが生まれることを示したかったのです．それに，自分でプログラムを書くのが快適な言語を開発したかったということもあります．

Bruce　Scala のどの部分がご自身で一番気に入っていますか?

Dr. Odersky　プログラマが自分のアイデアを自由に表現できる点，軽量でありながら型システムによって強力なサポートも得られる点が気に入っています．

Bruce　Scala はどのような問題の解決に最も有用ですか?

Dr. Odersky　Scala は，本当に汎用性の高い言語です．個人的には，Scala を使って解決しようと思わない問題はありません．それはそれとして，他の主要言語にはない Scala 固有の強みは，関数型プログラミングをサポートしている点です．したがって，並行性，XML を扱う Web アプリケーション，ドメイン固有言語の実装など，関数型のアプローチが肝要となる場面で，Scala は威力を発揮します．

Bruce　最初から開発をやり直せるとしたら，どの機能を変更したいですか?

Dr. Odersky　Scala 固有の型推論は一般にはうまく動作しますが，制限があります．やり直せるとしたら，より強力な制約ソルバーを用いるでしょう．これは今からでも変更可能かもしれませんが，大規模なインストールベースに対処する必要があるため難しくなります．

Twitter が，中核となるメッセージ処理の実装言語を Ruby から Scala に切り替えたため，Scala を取り巻くさまざまな噂が広がっている．Scala はオブジェクト指向機能を備えているため，Java からの移行は極めてスムーズだが，注目を集めているのは Scala の関数型プログラミング機能のほうだ．純粋な関数型言語を使うと，強力な数学的基盤のもとでプログラミングスタイルを実現できる．関数型言語の主要な特徴は次の 3 つである．

- 関数型プログラムは関数で構成される．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- 関数は常に値を返す．
- 関数は，与えられた入力が同じなら，常に同じ値を返す．
- 関数型プログラムは状態の変更やデータの変化を許さない．いったん設定した値はそのままにしておく必要がある．

厳密に言うと，Scala は純粋な関数型プログラミング言語ではない．これは C++ が純粋なオブジェクト指向言語でないのと同じだ．Scala は変更可能な値を許すので，同じ入力を与えたにもかかわらず異なる出力を返す関数を書くことができてしまう（大半のオブジェクト指向言語では，getter と setter を使うことでこのルールを破っている）．それでも，必要な場所で開発者が関数による抽象化を利用できるように，さまざまなツールが用意されている．

関数型プログラミングと並行性

並行性に関心のあるオブジェクト指向言語プログラマにとっての大きな問題は，可変状態，すなわち，データが変更可能であることだ．初期化後に別の値になりえる変数は，変更可能（mutable）であるといわれる．並行性と可変状態は，さしずめオースティン・パワーズと Dr. イーブルのような宿敵関係にある[*2]．2 つの異なるスレッドが同じデータを同時に変更できる場合，スレッドの実行後にデータが有効な状態になっていることを保証するのは難しい．有効な状態かどうかを判定するのはほぼ不可能だ．データベースシステムでは，トランザクションとロックを用いてこの問題に対処している．オブジェクト指向言語では，共有データに対するアクセスを制御するためのツールを用意することで対応している．ただし，プログラマはそうしたツールの使い方を知っていても，うまく使いこなせていないことが多い．

関数型プログラミング言語では，プログラミングの過程から可変状態を排除することで，この問題を解決できる．Scala では，可変状態を完全に排除する必要はないが，純粋に関数型スタイルでコーディングするためのツールは用意されている．

Scala を使えば，一部を Smalltalk にしようか，少しだけ Lisp で書こうかなどと迷う必要はない．では実際に Scala のコードを書いて，オブジェクト指向の世界と関数型の世界を融合させてみよう．

5.2　1 日目：丘の上の城

映画『シザーハンズ』では，丘の上にちょっと普通とは違った城が登場する．昔はこの城も不思議で魅力的な場所だったのだろうが，今ではすっかり古びてしまい，無視されているような感じだ．壊れた窓から雨風が吹き込み，部屋はすっかりすさんでしまい昔の面影はまったくない．かつては快適な住み処だったが，今では冷たく誰も近づかない．オブジェクト指向パラダイムはこの丘の上の城に似ている．とりわけ初期のオブジェクト指向の実装は古くささを感じさせる．Java などは，静的型付けと並行性の実装が時代

[*2] ［訳注］『オースティン・パワーズ』（*Austin Powers*）．Jay Roach 監督．1997 年．60 年代に冷凍保存されたオースティンが，現代によみがえり，宿敵 Dr. イーブルと対決するコメディ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

遅れになっているため，モデルチェンジが必要になってきている．このセクションでは，この丘の上の城，つまりオブジェクト指向パラダイムの観点から Scala を眺めてみることにする．

Scala は Java 仮想マシン（JVM）上で動作する．ここでは Java の詳細については触れない．Java に関する情報はいたるところで無償で提供されているのでそちらを参照してほしい．これから学習を進めていくと気づくように，Scala には Java 的な考え方が取り込まれているが，この章ではそうした Java からの影響を受けている部分をできるだけ避けて，2 つの言語を同時に覚える必要がないようにする．ではさっそく，Scala をインストールしてみよう．本書ではバージョン 2.7.7 を使う．

Scala の型

Scala の準備ができたら，scala と入力してコンソールを開いてほしい．正常に動作していれば，何もエラーが表示されず，scala>プロンプトが表示される．ここで，次のようなちょっとしたコードを入力してみる．

```
scala> println("Hello, surreal world")
Hello, surreal world

scala> 1 + 1
res8: Int = 2

scala> (1).+(1)
res9: Int = 2

scala> 5 + 4 * 3
res10: Int = 17

scala> 5.+(4.*(3))
res11: Double = 17.0

scala> (5).+((4).*(3))
res12: Int = 17
```

見てのとおり，整数はオブジェクトだ．Java では，Int（プリミティブ型）と Integer（オブジェクト）の間の変換にさんざん苦労させられた．Scala では，若干の例外はあるものの，すべてがオブジェクトだ．これは，大半の静的に型付けされたオブジェクト指向言語から見ると大きな進歩だ．Scala での文字列の扱いを見てみよう．

```
scala> "abc".size
res13: Int = 3
```

文字列もやはり，（若干のシンタックスシュガーは含まれているものの）ファーストクラスのオブジェクトだ．型の不一致を発生させてみよう．

```
scala> "abc" + 4
res14: java.lang.String = abc4

scala> 4 + "abc"
res15: java.lang.String = 4abc
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
scala> 4 + "1.0"
res16: java.lang.String = 41.0
```

うーむ．少しこちらの予想と違っていたようだ．Scala では，整数を文字列に強制的に変換している．もう少しあからさまな型の不一致だとどうなるだろう．

```
scala> 4 * "abc"
<console>:5: error: overloaded method value * with alternatives (Double)Double
<and> (Float)Float <and> (Long)Long <and> (Int)Int <and> (Char)Int
<and> (Short)Int <and> (Byte)Int cannot be applied to (java.lang.String)
       4 * "abc"
         ^
```

そうそう，これだ．Scala は，実は強く型付けされた言語なのだ．Scala では型推論によってほとんどの変数の型をシンタックスを手がかりに突き止めることができる．ただし Ruby と違って，この型チェックはコンパイル時に実行される．Scala のコンソールは，コードをコンパイルしては少しずつ実行している．

上の説明を読んで，Java の文字列の説明と同じだと感じたことと思う．Scala に関する大半の記事や書籍では，こうした Java との統合性を詳しく説明しているが，本書では紙幅の都合上，皆さんにとって最も興味深い部分であろうプログラミング構文にこのまま入っていくことにする．解説の中で，Java との統合について詳しく解説した書籍を随時紹介する．現時点では，2 つの言語にまたがって型を管理するための戦略がいたるところに用意されているとだけ述べておく．道理にかなった箇所では，`java.lang.String` のような単純な Java の型を使うというのもそうした戦略の 1 つだ．少し解説を簡略化しすぎと思われるかもしれないが，どうか筆者を信じて読み進めてほしい．

式と条件

それでは例を用いて，簡単ではあるが厳密に，基本的なシンタックスについて見ていくことにしよう．以下に Scala の真偽式をいくつか示す．

```
scala> 5 < 6
res27: Boolean = true

scala> 5 <= 6
res28: Boolean = true

scala> 5 <= 2
res29: Boolean = false

scala> 5 >= 2
res30: Boolean = true

scala> 5 != 2
res31: Boolean = true
```

ここでも面白いことは何もない．ここまでに紹介した言語で既におなじみの C 形式のシンタックスだ．`if` 文で式を使ってみよう．

```
scala> val a = 1
a: Int = 1
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
scala> val b = 2
b: Int = 2

scala> if ( b < a) {
     |   println("true")
     | } else {
     |   println("false")
     | }
false
```

いくつかの変数に値を代入し，それらを if/else 文で比較している．変数に代入している箇所をよく見てほしい．まず，型を指定していない．Ruby と違って Scala では，型をコンパイル時にバインド（束縛）する．ただし Java と違って Scala は型を推論できるため，val a : Int = 1 などと書く必要はない（書いても構わないが）．

次に，変数の宣言が val キーワードで始まっている．var キーワードも使える．val で宣言した変数は変更不能だが，var で宣言した変数は変更可能だ．この点については後述する．

Ruby では，0 は true に評価される．C では，0 は false に評価される．また，Ruby でも C でも，nil は false に評価される．Scala ではどうだろう．見てみよう．

```
scala> Nil
res3: Nil.type = List()

scala> if(0) {println("true")}
<console>:5: error: type mismatch;
 found   : Int(0)
 required: Boolean
       if(0) {println("true")}
          ^

scala> if(Nil) {println("true")}
<console>:5: error: type mismatch;
 found   : Nil.type (with underlying type object Nil)
 required: Boolean
       if(Nil) {println("true")}
          ^
```

見てのとおり，Nil は空のリストだ．また，Nil や 0 の真偽を判定することはできない．この振る舞いは，強く型付けされた静的言語である Scala の考え方と一致する．Nil および数値はブール値ではないため，ブール値として扱ってはならない．簡単な式と最も基本的な判定構文について見たところで，ループの説明に移ろう．

静的型付けとの筆者の内なる戦い

プログラミング言語おたくになって日が浅い人のなかには，強い型付けと静的な型付けを混同している人がいる．おおざっぱに言うと，強い型付けとは，2 つの型に互換性があるかどうかを検出し，互換性がなければ，エラーを投げるか，型の強制変換を行うという意味だ．表面的には，Java と Ruby はどちらも強く型付けされ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

た言語である（もちろんこれは単純化のしすぎであることは分かっている）．一方，アセンブリ言語やCのコンパイラは，弱く型付けされている．これらのコンパイラは，特定のメモリ位置にあるデータが整数なのか，文字列なのか，あるいは単なるデータなのかを必ずしも意識する必要はない．

静的な型付けと動的な型付けはまた別の問題だ．静的に型付けされた言語では，型の構造体に基づいてポリモーフィズムが行われる．つまり，遺伝子的な青写真によってアヒルかどうかを判定するのが静的型付け，鳴き声や歩き方によってアヒルかどうかを判定するのが動的型付けだ．静的に型付けされた言語は，コンパイラとツールがコードに関するさまざまな情報を保持していて，エラーを捕捉したり，コードを強調表示したり，リファクタリングしたりできるという点では有利だが，そのために作業量が増え，制約が生まれるというコストが伴う．このトレードオフをどのように感じるかは，多くの場合，開発者としての経験の多寡によって左右される．

筆者のオブジェクト指向言語での開発経験はJavaで始まった．その後，Javaの静的型付けの呪縛から逃れようとさまざまなフレームワークが次から次へと登場した．IT業界では，Enterprise Java Beansの3つのバージョン（Spring，Hibernate，JBoss）や，特定の使用モデルの柔軟性を高めるアスペクト指向プログラミングに何億ドルも投資が行われた．Javaの型付けモデルを動的なものにしようとしてきたわけだが，その途中のあらゆる段階で熾烈な争いが繰り広げられてきた．そのありさまは，プログラミング環境というより，敵対する新興宗教間の争いという感じだった．筆者の著書も，動的フレームワークから動的言語までという，同じ道を辿っている．

こうしたJavaの各種フレームワーク間の争いのせいで，筆者の静的型付けに対する偏見が形作られた．この偏見は，Haskellとその素晴らしい静的型システムによって解消されてきてはいるが，まだ時間がかかりそうだ．筆者は心にやましいところはないとはいえ，そういうわけなので，できる限り平易でバイアスのかかっていない説明を心がけたい．

ループ

これから紹介するプログラムは少し複雑になるので，コンソールから入力するのではなく，スクリプトとして実行することにする．Scalaのスクリプトを実行するには，RubyやIoのスクリプトと同様，`scala path/to/program.scala`のように入力する．

結果セットに対して繰り返し処理を実行する方法については，2日目にコードブロックが出てきたときに紹介する．ここでは，ループ処理に絶対に必要なプログラミングスタイルだけに絞って説明することにする．説明を読めば分かってもらえると思うが，Java形式のループ処理構造に大変よく似ている．

最初に，基本的な`while`ループを示す．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:50:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/scala/while.scala

```
def whileLoop {
    var i = 1
    while(i <= 3) {
        println(i)
        i += 1
    }
}

whileLoop
```

この例では関数を定義した．余談だが，Javaの開発者であれば，publicを付けなくてよいことにお気づきかと思う．Scalaではpublicがデフォルトの可視性になっている．つまりこの関数はすべてのユーザから見える．

メソッド内では，3まで数える簡単なwhileループを宣言する．iは変化するため，varで宣言している．次にJava形式のwhile文がくる．お分かりのとおり，条件がfalseになるまでブレース内のコードが実行される．このコードを実行するには次のように入力する．

```
batate$ scala code/scala/while.scala
1
2
3
```

forループは，JavaやCの場合とほとんど同じだが，構文が若干異なる．

▶ code/scala/for_loop.scala

```
def forLoop {
    println( "for loop using Java-style iteration" )
    for(i <- 0 until args.length) {
        println(args(i))
    }
}

forLoop
```

引数は，変数，<-演算子，ループの範囲（開始値 until 終了値）の順に指定する．この例ではコマンドラインの各引数をループで処理している．

```
batate$ scala code/scala/forLoop.scala its all in the grind
for loop using Java-style iteration
its
all
in
the
grind
```

Rubyと同様にコレクションをループで回すこともできる．ここではforeachを紹介する（これはRubyのeachを連想させる）．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/scala/ruby_for_loop.scala

```
def rubyStyleForLoop {
    println( "for loop using Ruby-style iteration" )
    args.foreach { arg =>
        println(arg)
    }
}

rubyStyleForLoop
```

args は，入力コマンドライン引数のリストだ．Scala は，このブロックに各要素を 1 つずつ渡す．この例では arg が，入力引数リスト args から取り出した 1 つの引数となる．Ruby で同じコードを書くと，args.each {|arg| println(arg) }のようになる．各引数を指定する構文は少し違っているが，考え方は同じだ．上のコードを実行すると次のような結果になる．

```
batate$ scala code/scala/ruby_for_loop.scala freeze those knees chickadees
for loop using Ruby-style iteration
freeze
those
knees
chickadees
```

後で，命令型のループよりも，この繰り返し方法のほうを頻繁に使うことに気づくと思うが，ここでは目の前のことに集中し，構文に関する議論は後回しにする．

範囲とタプル

Ruby 同様，Scala は，ファーストクラスの範囲をサポートしている．コンソールを起動して，次の簡単なコードを入力してみてほしい．

```
scala> val range = 0 until 10
range: Range = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> range.start
res2: Int = 0

scala> range.end
res3: Int = 10
```

これは問題ないだろう．Ruby の範囲と同じように動作する．増分を指定することもできる．

```
scala> range.step
res4: Int = 1


scala> (0 to 10) by 5
res6: Range = Range(0, 5, 10)

scala> (0 to 10) by 6
res7: Range = Range(0, 6)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Rubyの範囲1..10は1 to 10，Rubyの範囲1...10は1 until 10となる．境界を含める場合は，toを用いる．

```
scala> (0 until 10 by 5)
res0: Range = Range(0, 5)
```

範囲の向きも指定できる．

```
scala> val range = (10 until 0) by -1
range: Range = Range(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

ただし，向きは両辺の値からは推論されない．

```
scala> val range = (10 until 0)
range: Range = Range()

scala> val range = (0 to 10)
range: Range.Inclusive = Range(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

範囲の終点に関係なく，デフォルトの増分は1になる．範囲は整数に限らない．

```
scala> val range = 'a' to 'e'
range: RandomAccessSeq.Projection[Char] = RandomAccessSeq.Projection(a, b, c, d, e)
```

Scalaは，暗黙の型変換を行う．実際，for文を指定したとき，実は，範囲を指定していたのである．

Prolog同様，Scalaにはタプルが用意されている．タプルとは固定長のオブジェクト・セットだ．このパターンは，他の多くの関数型言語でも採用されている．タプル内のオブジェクトは型が異なっていても構わない．純粋な関数型言語ではオブジェクトとその属性をタプルで表すことが多い．次の例を見てほしい．

```
scala> val person = ("Elvis", "Presley")
person: (java.lang.String, java.lang.String) = (Elvis,Presley)

scala> person._1
res9: java.lang.String = Elvis

scala> person._2
res10: java.lang.String = Presley

scala> person._3
<console>:6: error: value _3 is not a member of (java.lang.String, java.lang.String)
       person._3
              ^
```

Scalaでは，多値代入にリストではなくタプルを使う．

```
scala> val (x, y) = (1, 2)
x: Int = 1
y: Int = 2
```

タプルは固定長なので，各タプル値に基づいて静的な型チェックを実行できる．

```
scala> val (a, b) = (1, 2, 3)
<console>:15: error: constructor cannot be instantiated to expected type;
 found   : (T1, T2)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
 required: (Int, Int, Int)
       val (a, b) = (1, 2, 3)
           ^
<console>:15: error: recursive value x$1 needs type
       val (a, b) = (1, 2, 3)
            ^
```

以上の基本事項を踏まえたうえで，それらすべてを組み合わせて，オブジェクト指向のクラス定義を作成してみよう．

Scala のクラス

最も単純なクラス，すなわち属性だけでメソッドもコンストラクタもないクラスは，Scala では次のように 1 行で簡単に定義できる．

```
class Person(firstName: String, lastName: String)
```

単純な値クラスを指定するのに本体を指定する必要はない．Person クラスは public となり，firstName 属性と lastName 属性をもつ．このクラスは，次のようにしてコンソールで使うことができる．

```
scala> class Person(firstName: String, lastName: String)
defined class Person

scala> val gump = new Person("Forrest", "Gump")
gump: Person = Person@7c6d75b6
```

もう少し複雑にしてみる．オブジェクト指向クラスでは，データと振る舞いを混ぜ合わせる．Scala で，完全なオブジェクト指向クラスを構築してみよう．このクラスを Compass と呼ぶことにする．コンパスの方位は最初，北向きになっている．コンパスに左または右に 90 度回転するように指示し，それに従って方位を変更する．これを実現する Scala のコード（全体）は次のようになる．

▶ code/scala/compass.scala

```
class Compass {

  val directions = List("north", "east", "south", "west")
  var bearing = 0

  print("Initial bearing: ")
  println(direction)

  def direction() = directions(bearing)

  def inform(turnDirection: String) {
    println("Turning " + turnDirection + ". Now bearing " + direction)
  }

  def turnRight() {
    bearing = (bearing + 1) % directions.size
    inform("right")
  }
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
  def turnLeft() {
    bearing = (bearing + (directions.size - 1)) % directions.size
    inform("left")
  }
}

val myCompass = new Compass

myCompass.turnRight
myCompass.turnRight

myCompass.turnLeft
myCompass.turnLeft
myCompass.turnLeft
```

構文は比較的簡単だが，いくつか風変わりな部分がある．コンストラクタは，インスタンス変数（コンストラクタに渡すインスタンス変数を除く）とメソッドを定義する．Ruby と違って，すべてのメソッド定義には引数の型と名前を指定する．また，コードの最初のブロックは，どのメソッド定義にも含める必要はない．少しずつ説明しよう．

▶ code/scala/compass.scala

```
class Compass {

  val directions = List("north", "east", "south", "west")
  var bearing = 0

  print("Initial bearing: ")
  println(direction)
```

クラス定義の後のコードブロック全体が実はコンストラクタになっている．コンストラクタには，方位と目標物の方角（ベアリング．方位のインデックス）の List が指定されている．あとで，体を回転させる（コンパスの向きを変える）とベアリングが変わる．次に，クラスの利用者に現在の方角を（英語で）示すための便利なメソッドがいくつか用意されている．

▶ code/scala/compass.scala

```
def direction() = directions(bearing)

def inform(turnDirection: String) {
  println("Turning " + turnDirection + ". Now bearing " + direction)
}
```

その後にメソッド定義が続く．direction メソッドは単純に，インデックスが bearing の directions の要素を返す．Scala には，1 行メソッドを定義する便利な代替シンタックスが用意されている．このシンタックスを使うと，メソッド本体を囲むブレースを省略できる．

inform メソッドは，利用者が体を回転させるたびに，その旨を知らせるメッセージを出力する．このメソッドは，回転の方向を引数として受け取り，値は返さない．では，回転を処理するメソッドの中身を見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/scala/compass.scala

```
def turnRight() {
  bearing = (bearing + 1) % directions.size
  inform("right")
}

def turnLeft() {
  bearing = (bearing + (directions.size - 1)) % directions.size
  inform("left")
}
```

turns メソッドは，回転の方向に基づいてベアリングを変える．% 演算子は，剰余算を行う（除算を実行し，その商は破棄して余りのみを返す）．右を向くとベアリングに 1 が加算され，左を向くと 1 が減算される．結果は方向に応じてラッピングされる．

補助コンストラクタ

以上，基本的なコンストラクタの挙動を説明した．コンストラクタはクラスとメソッドを初期化するコードブロックだ．別のコンストラクタを作成することもできる．次の Person クラスには 2 つのコンストラクタがある．

▶ code/scala/constructor.scala

```
class Person(first_name: String) {
  println("Outer constructor")
  def this(first_name: String, last_name: String) {
    this(first_name)
    println("Inner constructor")
  }
  def talk() = println("Hi")

}

val bob = new Person("Bob")
val bobTate = new Person("Bob", "Tate")
```

このクラスには，1 つの引数 firstName とメソッド talk が指定されたコンストラクタがある．this メソッドに注意してほしい．これが二番目のコンストラクタだ．このメソッドは，2 つの引数 firstName と lastName を受け取り，this をプライマリコンストラクタとして呼び出す．その際，引数として firstName だけを渡す．

クラス定義の後のコードは，Person を 2 つの方法で（最初はプライマリコンストラクタを用いて，次に補助コンストラクタを用いて）インスタンス化している．

```
batate$ scala code/scala/constructor.scala
Outer constructor
Outer constructor
Inner constructor
```

説明は以上で終わりだ．補助コンストラクタはさまざまな使い方ができるので重要だ．クラスメソッドの作成方法を見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

クラスの拡張

ここまで扱ってきたクラスはごく基本的なものだ．属性とメソッドしか持たないごく基本的なクラスをいくつか作成した．このセクションでは，クラスの相互作用について見ていく．

コンパニオンオブジェクトとクラスメソッド

Java と Ruby では，同じコード本体内にクラスメソッドとインスタンスメソッドを両方作成する．Java では，クラスメソッドに static キーワードを付ける．Ruby では，def self.class_method という構文を用いる．Scala ではどちらの方法も用いない．その代わり，class 定義内にインスタンスメソッドを宣言する．インスタンスを 1 つしか持つことができないものがある場合は，class キーワードではなく object キーワードを用いて定義する．以下に例を示す．

▶ code/scala/ring.scala

```
object TrueRing {
 def rule = println("To rule them all")
}

TrueRing.rule
```

TrueRing の定義は他の class 定義とまったく同じように動作するが，シングルトンオブジェクト（Ruby でいう特異オブジェクト）を生成する点が異なる．Scala では，object 定義と class 定義に同じ名前を使うことができる．その場合，シングルトンオブジェクト宣言内にクラスメソッドに相当するものを定義し，クラス宣言内にインスタンスメソッドを定義する．上の例では，rule メソッドがクラスメソッドに相当する．また同名のクラスを持つそのようなオブジェクトをコンパニオンオブジェクトと呼ぶ．

継承

Scala における継承は極めて単純だが，構文は正確でなければならない．以下に，Person クラスを Employee クラスで拡張する例を示す．Employee では，id フィールドに社員番号を格納できるようになっている．次のコードを見てほしい．

▶ code/scala/employee.scala

```
class Person(val name: String) {
  def talk(message: String) = println(name + " says " + message)
  def id(): String = name
}


class Employee(override val name: String,
                        val number: Int) extends Person(name) {
  override def talk(message: String) {
    println(name + " with number " + number + " says " + message)
  }
  override def id():String = number.toString
}
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
val employee = new Employee("Yoda", 4)
employee.talk("Extend or extend not. There is no try.")
```

この例では，Person 基底クラスを Employee で拡張している．Employee に新しいインスタンス変数 number を追加し，talk メッセージをオーバーライドして，新しい振る舞いを追加している．分かりにくい構文の多くは，クラスのコンストラクタ定義に関するものだ．Person には完全な引数リストを指定する必要があるが，型は省略できる．

override キーワードは，コンストラクタ内でも，基底クラスから拡張するメソッドにも，必須である．このキーワードを指定することで，綴りを間違えて新しいメソッドを定義してしまうのを防ぐことができる．全般に，この辺りで驚くようなことは何もないが，個人的には，ときどき，エドワードが幼気な赤ちゃんうさぎをはさみの手で可愛がろうとしているような感じを受けるときがある．次に進もう．

トレイト

すべてのオブジェクト指向言語は，1 つのオブジェクトが複数の異なる役割を担うことがあるという問題を解決しなければならない．ここに永続化可能なオブジェクトがあったとしよう．バイナリデータを MySQL にプッシュする方法をこのオブジェクトが逐一知らなければならないのでは大変だ．そこで C++ は多重継承を，Java はインタフェースを，Ruby は mixin を，そして Scala はトレイト（trait）[*3]を使う．Scala のトレイトは Ruby の mixin と同様，モジュール内に実装する．あるいは，Java のインタフェースに実装を加えたものとも解釈できる．クラスを部分的に実装したものと考えればよい．理想的には，単一の関心事を実装するのに使うのがよい．以下に，Nice というトレイトを Person に追加する例を示す．

▶ code/scala/nice.scala

```
class Person(val name:String)

trait Nice {
  def greet() = println("Howdily doodily.")
}

class Character(override val name:String) extends Person(name) with Nice

val flanders = new Character("Ned")
flanders.greet
```

最初の要素は Person だ．Person は name という単一の属性を持つ簡単な class だ．2 番目の要素が Nice という名前のトレイトだ．これは mixin であり，greet という名前の単一のメソッドを持っている．最後の要素は，Character という名前のクラスであり，Nice トレイトを mixin している．これで，Character の任意のインスタンスに対して greet メソッドを呼ぶことができる．出力結果は次のように期待どおりになる．

```
batate$ scala code/scala/nice.scala
```

[*3] ［訳注］英語での一般的な意味は特徴，特質といったところ．トレイトという仕組みを採用したプログラミング言語は Self が最初だろう．現在では Smalltalk をはじめ多くの言語で採用されている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Howdily doodily.
```

ここは複雑なことは何もない．greet というメソッドが定義されたトレイト Nice を，Scala の任意のクラスに mixin して，greet の振る舞いを取り込んでいる．

1日目で学んだこと

1日目では，1つの言語で2つの異なるプログラミングパラダイムを説明する必要があるため，大変広い範囲にわたって説明を行った．Scala はオブジェクト指向の概念を取り込んだ言語であり，JVM（Java 仮想マシン）上で既存の Java ライブラリと連携して実行される．Scala のシンタックスは Java のシンタックスに似ており，静的で強く型付けされている．Martin Odersky は，2つのパラダイム，すなわちオブジェクト指向プログラミングと関数型プログラミングの橋渡しをするために Scala を開発した．2日目で紹介する関数型プログラミングの概念により，並行アプリケーションの設計が容易になる．

Scala の静的型付けでは，型の推論も行われる．ユーザは，すべての状況ですべての変数の型を常に宣言する必要はない．Scala は，シンタックスを手がかりに型を推論できることが多いからだ．また，コンパイラも整数から文字列という具合に型の強制変換を実行できるので，意味をなす場合には，暗黙の型変換が可能である．

Scala の式は他の言語の式とほとんど同じように動作するが，他の言語と比べて少しばかり厳密である．ほとんどの条件文にはブール型を指定する必要があるが，0 や Nil をブール値として使うことはできない．これらは true にも false にもならない．ループ構文や制御構文は他の言語とほとんど変わらない．また Scala では，タプル（異なる型を保持できる固定長のリスト）や範囲（固定長の境界値を含む順序付き数列）などの高度な型もサポートされている．

Scala のクラスは Java のクラスとほぼ同じだ．ただし，Scala では，クラスメソッドはサポートされておらず，代わりに，コンパニオンオブジェクトという概念を用いて，同じクラス上にクラスメソッドとインスタンスメソッドを混在させることができる．Ruby では mixin を，Java ではインタフェースを使用するが，Scala ではトレイトと呼ばれる mixin と同じような構造体を使用する．

2日目では，Scala の関数型機能についてひととおり説明する．コードブロック，コレクション，変更不能な変数，および foldLeft などの高度な組み込みメソッドについても説明する．

セルフスタディ1日目

1日目の解説は広範囲にわたったが，大半は，おなじみのトピックだったはずだ．これらのオブジェクト指向の概念は誰でも知っているに違いない．この演習は，これまでの章の演習に比べると少し難易度が高いが，解けるはずだ．

探してみよう：

- Scala API
- Java と Scala の比較
- val と var を比較した説明

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

試してみよう：

- X，O，および空白文字で埋まった三目並べボードを見て，勝者，引き分け，勝者未定のいずれかの判定を下すゲームを書け．必要ならクラスを使うこと．
- ボーナス問題: 2 人のプレーヤーに三目並べを行わせてみよ．

5.3　2 日目：藪を刈り込む

映画『シザーハンズ』では，エドワードが「自分は人里離れた丘の上の城からやってきたが，特殊な能力のおかげで普通の人間社会でも特別な居場所があるのではないか」ということを理解したときに魔法のような瞬間が訪れる．

プログラミング言語の歴史にある程度の知識のある人なら誰でも，この嘘のような話が現実に起きたことを知っている．オブジェクト指向パラダイムがまだ新しい概念だったとき，一般のプログラマ達は，パラダイムが新しすぎるが故に Smalltalk を受け入れることができなかった．我々が求めていたのは，手続き型プログラミングをそのまま維持しながら，オブジェクト指向の概念を実験できる，そんな言語だった．C++ の登場で，新しいオブジェクト指向の技法を，C による既存の手続き型機能と安全に共存させることができるようになった．その結果，プログラマ達は，それまでの古いコンテキストで新しい技法を試すことができるようになったのである．

さて，そろそろ関数型言語としての Scala の力量を試してみることにしよう．関数型言語としての Scala の機能のなかには，最初はぎこちなく思えるものもあるだろうが，その概念は強力かつ重要である．そして，それは 3 日目で説明する並行構文の基盤を形成することになる．では，まず簡単な関数から始めよう．

```
scala> def double(x:Int):Int = x * 2
double: (Int)Int

scala> double(4)
res0: Int = 8
```

関数の定義は，Ruby の場合とほとんど変わらない．def キーワードで，関数とメソッドの両方を定義する．その後に，引数とその型を指定する．さらにその後に，戻り型（省略可能）を指定できる．Scala では多くの場合，戻り型を推論できる．

関数を呼び出すには，関数名と引数のリストを指定する．その際，Ruby と違って，括弧は省略できない．

上記の例は 1 行のメソッド定義である．次のように，メソッドをブロック形式で指定することもできる．

```
scala> def double(x:Int):Int = {
     |   x * 2
     | }
double: (Int)Int

scala> double(6)
res3: Int = 12
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:51:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Int 戻り型の後の=は必須だ．これを忘れると面倒なことになる．以上が関数宣言の主要な形式だ．引数を省略するなど，少し変則的な形式もあるが，これらの形式が最もよく使われる．

では関数内で使う変数の説明に移ろう．純粋な関数型プログラミングモデルについて学ぶなら，変数のライフサイクルに特に注意を払う必要がある．

var 対 val

Scala は Java の仮想マシンを基盤としており，Java と緊密な関係がある．こうした設計上の目標は，ある意味では Scala という言語を制限しているが，別の観点から見れば，過去 15〜20 年にわたるプログラミング言語の発展（の成果）を活かすことができるとも言える．学習を進めていくと，Scala では並行プログラミングとの親和性がかなり重視されていることに気づくだろう．しかし，基本的な設計原理に従わなければ，どのような並行性機能も役に立たない．可変状態は有害だ．いったん宣言した変数は，衝突状態を避けるために変更不能（immutable）にすべきである．Java ではそのために final キーワードを使う．Scala では，var の代わりに val を使うことで，変更不能であることを宣言する．

```
scala> var mutable = "I am mutable"
mutable: java.lang.String = I am mutable

scala> mutable = "Touch me, change me..."
mutable: java.lang.String = Touch me, change me...

scala> val immutable = "I am not mutable"
immutable: java.lang.String = I am not mutable

scala> immutable = "Can't touch this"
<console>:5: error: reassignment to val
       immutable = "Can't touch this"
                 ^
```

ご覧のとおり，var で宣言した値は変更可能だが，val で宣言した値は変更不能だ．コンソールでは便宜上，val 定義であっても再定義が許されているが，通常は val の内容を再定義するとエラーになる．

ある意味，Scala では，従来の命令型のプログラミングスタイルをサポートするために var 形式の変数を導入する必要があったのだが，Scala を学習している間は，並行性を高めるため，var を使うのはできる限り避けたほうがよい．この「可変状態は並行性を制限する」という基本的な設計哲学は，関数型プログラミングとオブジェクト指向プログラミングを差別化する基本要素となっている．

それでは，関数型言語で筆者が気に入っている部分である各種コレクションの説明に入ることにしよう．

コレクション

関数型言語には，その長い歴史の中で開発された各種コレクションを扱うための非常に便利な機能がある．初期の関数型言語の 1 つである Lisp は，リスト処理の概念を骨格

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

として開発された言語である．Lisp という名前自体，LISt Processing の略である．関数型言語を使うと，データとコードを含む複雑な構造を簡単に構築できる．Scala に用意されている主なコレクションとして，リスト，セット，マップがある．

リスト

ほとんどの関数型言語と同様，基本的なデータ構造としてリストがある．Scala のリスト（型は `List`）は，同種のモノのランダムアクセス可能な順序付きコレクションだ．次のリストをコンソールに入力してみてほしい．

```
scala> List(1, 2, 3)
res4: List[Int] = List(1, 2, 3)
```

最初の戻り値 `List[Int] = List(1, 2, 3)` に注目してほしい．この値はリスト全体の型を示しているだけでなく，リスト内のデータ構造の型も示している．`String` のリストは次のようになる．

```
scala> List("one", "two", "three")
res5: List[java.lang.String] = List(one, two, three)
```

「少し Java の影響を感じるな」と思った方は正しい．Java には，リストや配列などのデータ構造体内の項目に型付けするための Generics という機能がある．`String` と `Int` をリスト内で混在させたらどうなるか見てみよう．

```
scala> List("one", "two", 3)
res6: List[Any] = List(one, two, 3)
```

データ型は `Any`（Scala の汎用データ型）になる．リスト内の項目にアクセスするには，次のように指定する．

```
scala> List("one", "two", 3)(2)
res7: Any = 3

scala> List("one", "two", 3)(4)
java.util.NoSuchElementException: head of empty list
        at scala.Nil$.head(List.scala:1365)
        at scala.Nil$.head(List.scala:1362)
        at scala.List.apply(List.scala:800)
        at .<init>(<console>:5)
        at .<clinit>(<console>)
        at RequestResult$.<init>(<console>:3)
        at RequestResult$.<clinit>(<console>)
        at RequestResult$result(<console>)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Met...
```

このように `()` 演算子を使う．リストへのアクセスは関数として実現されているため，`[]` ではなく `()` を使う．Scala では，リストのインデックスは 0 から始まる．この点は，Java や Ruby と同じだ．ただし，Ruby と違って，範囲外の項目にアクセスすると例外が発生する．負の数をインデックスに指定するとどうなるだろうか．次のように，初期のバージョンでは先頭の要素が返される．

```
scala> List("one", "two", 3)(-1)
res9: Any = one
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
scala> List("one", "two", 3)(-2)
res10: Any = one

scala> List("one", "two", 3)(-3)
res11: Any = one
```

この動作は，上限を超えるインデックスを指定すると NoSuchElement という例外が発生する動作と矛盾するため，バージョン 2.8.0 で java.lang.IndexOutOfBoundsException を返すように修正された．

最後に，Scala では，Nil は空のリストである点に注意してほしい．

```
scala> Nil
res33: Nil.type = List()
```

リストはコードブロックを説明するとき，基本構築ブロックとして使うが，その前に，リスト以外のコレクションについて紹介しておく．もう少し我慢してお付き合い願いたい．

セット

セットはリストに似ているが，明示的な順序を持たない点が異なる．セットを指定するには Set キーワードを使う．

```
scala> val animals = Set("lions", "tigers", "bears")
animals: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, tigers, bears)
```

セットに対する項目の追加と削除（セットに対する加算と減算）は簡単だ．

```
scala> animals + "armadillos"
res25: scala.collection.immutable.Set[java.lang.String] =
    Set(lions, tigers, bears, armadillos)

scala> animals - "tigers"
res26: scala.collection.immutable.Set[java.lang.String] = Set(lions, bears)

scala> animals + Set("armadillos", "raccoons")
<console>:6: error: type mismatch;
 found   : scala.collection.immutable.Set[java.lang.String]
 required: java.lang.String
       animals + Set("armadillos", "raccoons")
                 ^
```

セット演算は破壊的ではない点に注意してほしい．つまり，セット演算を実行するたびに新しいセットが作成される．既存のセットが変更されることはない．デフォルトではセットは変更不能になっている．お分かりのように，項目の追加や削除は極めて簡単だが，Ruby と違って，+ や - を用いてセット同士の演算はできない．Scala では，セットの和と差を求めるのに，それぞれ ++ と -- を用いる．

```
scala> animals ++ Set("armadillos", "raccoons")
res28: scala.collection.immutable.Set[java.lang.String] =
  Set(bears, tigers, armadillos, raccoons, lions)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
scala> animals -- Set("lions", "bears")
res29: scala.collection.immutable.Set[java.lang.String] = Set(tigers)
```

**を使ってセットの積（2 つのセット間に共通する要素）を求めることもできる[*4]:

```
scala> animals ** Set("armadillos", "raccoons", "lions", "tigers")
res1: scala.collection.immutable.Set[java.lang.String] = Set(lions, tigers)
```

リストと違って，セットは順序に依存しない．このため，セットとリストでは等価の意味が異なる．

```
scala> Set(1, 2, 3) == Set(3, 2, 1)
res36: Boolean = true

scala> List(1, 2, 3) == List(3, 2, 1)
res37: Boolean = false
```

これだけ知っておけば，とりあえずセットの操作は問題ないだろう．次にマップの説明に移ろう．

マップ

マップは Ruby の Hash と同じで，キーと値のペアだ．シンタックスは次のとおり．これは見たことがあるだろう．

```
scala> val ordinals = Map(0 -> "zero", 1 -> "one", 2 -> "two")
ordinals: scala.collection.immutable.Map[Int,java.lang.String] =
  Map(0 -> zero, 1 -> one, 2 -> two)

scala> ordinals(2)
res41: java.lang.String = two
```

リストやセット同様，Scala のマップを指定するには Map キーワードを使う．キーと値は->演算子で区切る．上の例では，Scala のマップを簡単に作成するためのシンタックスシュガーを使った．もう 1 つの形式であるハッシュマップを用いて，キーと値の型を指定してみよう．

```
scala> import scala.collection.mutable.HashMap
import scala.collection.mutable.HashMap

scala> val map = new HashMap[Int, String]
map: scala.collection.mutable.HashMap[Int,String] = Map()

scala> map += 4 -> "four"

scala> map += 8 -> "eight"

scala> map
res2: scala.collection.mutable.HashMap[Int,String] =
  Map(4 -> four, 8 -> eight)
```

[*4] **は廃止になるため，Scala 2.8.0 以降は&を使うこと．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

最初に，変更可能な HashMap を実現する Scala のライブラリをインポートする．つまり，このハッシュマップ内の値は変更できる．次に，map という名前の変更不能な変数を宣言する．つまり，マップに対する参照は変更できない．キーと値のペアの型を指定している点にも注意してほしい．最後に，キーと値のペアをいくつか追加して結果を返している．

間違った型を指定すると，次のようなことが起こる．

```
scala> map += "zero" -> 0
<console>:7: error: overloaded method value += with alternatives (Int)map.MapTo
    <and> ((Int, String))Unit cannot be applied to ((java.lang.String, Int))
       map += "zero" -> 0
           ^
```

予想どおり，型エラーが起きている．型エラーは，可能な場合はコンパイル時に検出されるが，実行時にも検出される．コレクションの基本的な説明は以上だ．次に，より詳細な説明に入ることにしよう．

Any と Nothing

無名関数の説明に入る前に，Scala のクラス階層についてもう少し説明しておこう．Scala と Java を併用する場合は，Java のクラス階層を意識することのほうが多いだろう．それでも，Scala の型について少し知っておく必要がある．Any は Scala クラス階層のルートクラスだ．混乱しがちだが，すべての Scala の型は Any を継承することを知っておいてほしい．

同様に，Nothing はすべての型の下位型（サブタイプ）である．このようにすることで，例えば，コレクション型の関数は Nothing を返して，特定の関数の戻り値に合わせることができる．これを図 5.1「Any と Nothing」(次ページ)に図示する．Any はすべての型の親であり，Nothing はすべての型の子である．

nil の概念を扱うときには若干ニュアンスの違いがある．Null はトレイトであり，null はそのインスタンスであって，空値を意味する Java の null と同じように動作する．空のコレクションは Nil である．対照的に，Nothing はトレイトであり，すべてのモノの下位型である．Nothing にはインスタンスがないため，Null と違って逆参照できない．例えば，メソッドが Exception を投げた場合，その戻り値は Nothing（つまり，値なし）である．

これらの規則を覚えておけば問題ないだろう．それでは高階関数を使って少し高度なコレクション操作を行ってみよう．

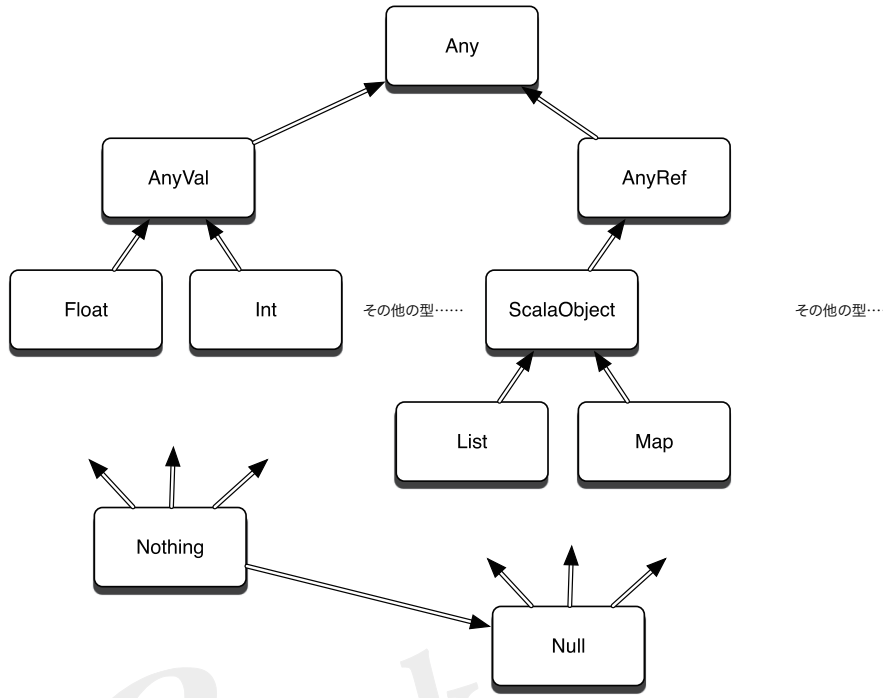
コレクションと関数

せっかく関数型言語の学習を始めたのだから，この辺りで，これまで説明してきた概念を改めて説明しておきたい．最初は高階関数だ．

Ruby や Io と同様，Scala のコレクションは，高階関数を使うことで格段に面白くなる．Ruby には each が，Io には foreach が用意されていた．Scala では，foreach に関数を渡すことができる．これらは，これまでの説明でも使ってきたが，その基盤となる概念が高階関数だ．やさしい言葉で言い換えれば，高階関数とは，関数を生産または消費する関

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 5.1　Any と Nothing

数である．具体的に言うと，高階関数とは，他の関数を入力引数として受け取る関数，または出力として関数を返す関数のことだ．このように他の関数を使用する関数を作成する手法は，関数型言語の極めて重要な概念であり，他の言語でのコーディングにも役に立つ．

Scala の高階関数は強力だ．本書では，部分適用関数やカリー化などの高度なトピックについて説明する余裕はないが，簡単な関数（コードブロックと呼ばれることが多い）をコレクションの引数として渡す方法については学習する．関数は任意の変数または引数に代入できる．それらを関数に渡すこともできるし，関数の戻り値として返すこともできる．ここでは，コレクションに定義されている面白いメソッドの入力引数として無名関数を渡す例を中心に説明する．

foreach

最初に取り上げる関数は `foreach` だ．`foreach` は Scala の繰り返し処理の中心となる．Io 同様，コレクションに対して `foreach` メソッドを呼ぶ際には，引数としてコードブロックを指定する．Scala では，このコードブロックを変数名 `=>` コードという形式で表

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

現する．次に例を示す．

```
scala> val list = List("frodo", "samwise", "pippin")
list: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.foreach(hobbit => println(hobbit))
frodo
samwise
pippin
```

hobbit => println(hobbit) は無名関数，つまり，名前のない関数だ．無名関数を宣言するには，=>の左側に引数を，右側にコードを指定する．foreach は無名関数を呼び出すが，その際，リストの各要素を入力引数として渡す．もうお分かりだと思うが，この手法は，セットやマップにも使える（ただし，順番は保証されない）．

```
val hobbits = Set("frodo", "samwise", "pippin")
hobbits: scala.collection.immutable.Set[java.lang.String] =
  Set(frodo, samwise, pippin)

scala> hobbits.foreach(hobbit => println(hobbit))
frodo
samwise
pippin

scala> val hobbits = Map("frodo" -> "hobbit",
  "samwise" -> "hobbit", "pippin" -> "hobbit")
hobbits: scala.collection.immutable.Map[java.lang.String,java.lang.String] =
  Map(frodo -> hobbit, samwise -> hobbit, pippin -> hobbit)

scala> hobbits.foreach(hobbit => println(hobbit))
(frodo,hobbit)
(samwise,hobbit)
(pippin,hobbit)
```

もちろんマップは要素の代わりにタプルを返す．思い出してほしいのだが，次のようにしてタプルの片方を参照することができる．

```
scala> hobbits.foreach(hobbit => println(hobbit._1))
frodo
samwise
pippin

scala> hobbits.foreach(hobbit => println(hobbit._2))
hobbit
hobbit
hobbit
```

これらの無名関数を使うと，単なる繰り返しよりはるかに高度な処理が可能だ．まずは基本的な手法をざっと説明し，その後で関数とコレクションを組み合わせた面白い手法をいくつか紹介することにしよう．

その他の List 操作用メソッド

ここで少し脇道に逸れるが，List のメソッドをもう少し紹介しておこう．これらの基本メソッドは，手動での繰り返しやリストの再帰処理に必要な機能を提供する．最初に，

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

リストが空かどうかをテストするメソッドやリストのサイズを確認するメソッドを紹介する．

```
scala> list
res23: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.isEmpty
res24: Boolean = false

scala> Nil.isEmpty
res25: Boolean = true

scala> list.length
res27: Int = 3

scala> list.size
res28: Int = 3
```

サイズの確認は，length と size のどちらでも実行できる点に注意してほしい．また，Nil は空リストとして実装されていることも覚えておくこと．Prolog の場合のように，リストの先頭と末尾を捕まえて再帰処理できるのは便利だ．

```
scala> list.head
res34: java.lang.String = frodo

scala> list.tail
res35: List[java.lang.String] = List(samwise, pippin)

scala> list.last
res36: java.lang.String = pippin

scala> list.init
res37: List[java.lang.String] = List(frodo, samwise)
```

少し驚いたかもしれない．先頭から先に再帰処理するには head と tail を使い，末尾から先に再帰処理するには last と init を使う．再帰については後でもう少し詳しく説明する．では，基本事項のまとめに，いくつか便利で面白いメソッドを紹介しておく．

```
scala> list.reverse
res29: List[java.lang.String] = List(pippin, samwise, frodo)

scala> list.drop(1)
res30: List[java.lang.String] = List(samwise, pippin)

scala> list
res31: List[java.lang.String] = List(frodo, samwise, pippin)

scala> list.drop(2)
res32: List[java.lang.String] = List(pippin)
```

これらのメソッドの動作は期待どおりだ．reverse はリストを逆順にして返す．drop(n) はリストの先頭の n 個の要素を削除して返す（元のリストは変更しない）．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

count，map，filter など

Ruby と同様，Scala にはリストに対してさまざまな操作を行う関数が多数用意されている．特定の条件を満たす要素を取り出す，任意の条件を用いてリストをソートする，各要素を入力として用い別のリストを作成する，要素値を合計するなどの操作が可能だ．

```
scala> val words = List("peg", "al", "bud", "kelly")
words: List[java.lang.String] = List(peg, al, bud, kelly)

scala> words.count(word => word.size > 2)
res43: Int = 3

scala> words.filter(word => word.size > 2)
res44: List[java.lang.String] = List(peg, bud, kelly)

scala> words.map(word => word.size)
res45: List[Int] = List(3, 2, 3, 5)

scala> words.forall(word => word.size > 1)
res46: Boolean = true

scala> words.exists(word => word.size > 4)
res47: Boolean = true

scala> words.exists(word => word.size > 5)
res48: Boolean = false
```

上の例を説明しよう．最初に Scala のリストを作成している．次にサイズが 2 を超える単語の数を数えている．count は，コードブロック word => word.size > 2 を呼び出し，リスト内の各要素について，word.size > 2 という式を評価し，true に評価された要素を数える．

同様に，words.filter(word => word.size > 2) は，サイズが 2 より大きいすべての単語のリストを返す．Ruby の select と同じだ．同じ要領で，map はリスト内のすべての単語のサイズのリストを作成し，forall はセット内のすべての項目がコードブロックに対して true を返したときに true を返し，exists は，セット内のいずれかの項目がコードブロックに対して true を返したときに true を返す．

コードブロックを用いて，より強力な処理が行えるように機能を一般化できる場合もある．例えば通常の文字列ソートは次のようになる．

```
scala> words.sort((s, t) => s.charAt(0).toLowerCase < t.charAt(0).toLowerCase)
res49: List[java.lang.String] = List(al, bud, kelly, peg)
```

このコードでは，2 つの引数 s と t を受け取るコードブロックを使っている．sort[*5]を使うと，2 つの引数を任意の方法で比較できる．上の例では，文字を小文字に変換し[*6]，両者を比較している．これにより，大文字と小文字を区別しない検索が行える．同じ方法で，リストを単語のサイズでソートすることもできる．

```
scala> words.sort((s, t) => s.size < t.size)
```

[*5] sort は，バージョン 2.8.0 で非推奨になった．代わりに sortWith を使うこと．
[*6] toLowerCase は，バージョン 2.8.0 で非推奨になった．代わりに toLower を使うこと．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
res50: List[java.lang.String] = List(al, bud, peg, kelly)
```

コードブロックを用いることで，望みのポリシーに基づいてソート[*7]を実行できる．もう少し複雑な例として，foldLeft を見てみよう．

foldLeft

Scala の foldLeft メソッドは，Ruby の inject メソッドによく似ている．foldLeft に初期値とコードブロックを与えると，配列の各要素ともう 1 つの値がコードブロックに渡される．このもう 1 つの値は，初期値（初回の呼び出し時）またはコードブロックからの戻り値（2 回目以降の呼び出し時）だ．このメソッドには 2 つのバージョンがある．最初のバージョン/:は，initialValue /: codeBlock という形式で使う演算子だ．実際の動作方法を以下に示す．

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val sum = (0 /: list) {(sum, i) => sum + i}
sum: Int = 6
```

この動作シーケンスは Ruby の章でも見たが，再度説明する．次のとおりだ．

- 値とコードブロックを与えて演算子を呼ぶ．コードブロックは 2 つの引数 sum と i を受け取る．
- 最初，/:は初期値 0 と list の先頭要素 1 を受け取り，それらをコードブロックに渡す．したがって sum が 0，i が 1 となり，0 + 1 が計算され，1 が結果として返される．
- 次に，/:は，コードブロックからの戻り値 1 を受け取り，それを sum に戻す（畳み込む）．したがって sum は 1 となる．i は list の次の要素，つまり 2 だ．コードブロックの実行結果は 3 となる．
- 最後に，/:が，コードブロックからの戻り値 3 を受け取り，それを sum に戻す．したがって sum は 3 となる．i は list の次の要素，つまり 3 だ．したがって sum + i は 6 となる．

foldLeft のもう 1 つのバージョンのシンタックスは奇妙に思えるかもしれない．カリー化と呼ばれる考え方を採用しているからだ．関数型言語では，カリー化を用いて，複数の引数を取る 1 つの関数を，それぞれ固有の引数リストを取る複数の関数に変換する．カリー化については第 8 章「Haskell」(p. 231)で詳しく説明する．ここでは，単一の関数ではなく，複数の関数の合成が行われているのだとだけ理解しておけばよい．2 つのバージョンでは，仕組みとシンタックスは異なるが，結果はまったく同じである．

```
scala> val list = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> list.foldLeft(0)((sum, value) => sum + value)
res54: Int = 6
```

[*7] sort は，バージョン 2.8.0 で非推奨になった．代わりに sortWith を使うこと．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:54:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

関数呼び出し `list.foldLeft(0)((sum, value) => sum + value)` は2つの引数リストがある点に注意してほしい．これが，先ほど説明したカリー化の概念だ．本書に登場するどの言語にも，このメソッドと同等のメソッドが用意されている．

2日目で学んだこと

1日目では，皆さん既にご存じのオブジェクト指向の機能をScalaで使う方法を説明した．2日目では，Scalaの最大の存在理由である関数型プログラミングを紹介した．

まず，基本的な関数から始めた．Scalaには，関数を定義するための柔軟なシンタックスが用意されている．戻り型はコンパイラで推論できることが多く，関数（またはメソッド）本体の定義には，1行形式とコードブロック形式があり，引数リストは可変長である．

次に，各種コレクションについて見た．Scalaでは，リスト，マップ，セットの3つのコレクションをサポートしている．セットはオブジェクトのコレクション，リストは順序付きのコレクション，そしてマップはキーと値のペアだ．Rubyと同様，コードブロックとコレクションのさまざまな強力な組み合わせについて説明した．関数型プログラミングパラダイムであることを感じさせるコレクションAPIについても見た．

リストでは，Prolog同様，Lisp形式の `head` および `tail` メソッドを用いて，リストの先頭要素と残りの要素を返すことができる．また，`count`，`empty`，`first` の各メソッド（機能は名前のとおり）も使える．最も強力なメソッドは関数ブロックを引数にとる．

`foreach` を用いた繰り返し処理，`filter` を用いたリスト内要素の絞り込みについても説明した．`foldLeft` を使って，コレクションの各要素に対して繰り返しながら計算結果を畳み込む（例えば中間合計を保持する）方法についても学んだ．

関数型プログラミングの大半は，Javaスタイルの繰り返しではなく，より高度な構文を用いてコレクションを操作する方法を学ぶことだ．3日目は，これらのスキルを実際に試してみる．具体的には，並行性の使用，XMLの処理，簡単な実用コードの作成について学ぶ．お楽しみに．

セルフスタディ2日目

Scalaについてかなり掘り下げて説明したので，Scalaの関数型言語としての側面が少し見えてきたはずだ．関数を扱うときは，コレクションから始めるのが一番だ．以下の演習では関数に加えてコレクションも使ってみる．

探してみよう：

- Scalaファイルの使い方の説明
- クロージャはコードブロックとどこが違うのか

試してみよう：

- `foldLeft` を用いて，文字列リストの合計長を計算せよ．
- 不敬語である *Shoot* と *Darn* を *Pucky* と *Beans* で置き換えるメソッドを持つ `Censor` トレイトを書け．マップを用いて不敬語とその代替語のペアを格納すること．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- 不敬語とその代替語をファイルから読み込むようにしてみよ．

5.4　3 日目：快刀乱麻を断つ

『シザーハンズ』のクライマックスシーンの直前でエドワードは，はさみの手を日常生活の中で芸術家並みに巧みに使うことを覚える．低木にはさみを入れて恐竜の形にしたり，ヴィダル・サスーンばりの手さばきでやすやすと見事な髪型を作り上げたり，焼いた肉を家族に切り分けてやったりする．Scala は，既に見たようにぎこちない部分もあるけれど，しっくりくるようになると，すごい威力を発揮する．XML や並行性などの難しい処理が，ほとんどありふれた処理と化してしまう．具体的に見ていこう．

XML

最近のプログラミングでは，XML 文書の処理を求められることがますます多くなってきた．Scala は，XML をファーストクラスの要素に昇格させるという劇的な進歩を遂げた．おかげで，文字列とまったく同じように XML を表現することができる．

```
scala> val movies =
     | <movies>
     |     <movie genre="action">Pirates of the Caribbean</movie>
     |     <movie genre="fairytale">Edward Scissorhands</movie>
     | </movies>
movies: scala.xml.Elem =
<movies>
          <movie genre="action">Pirates of the Caribbean</movie>
          <movie genre="fairytale">Edward Scissorhands</movie>
        </movies>
```

movies 変数を XML として定義すると，XML の各要素に直接アクセスできる．例えば，内部の本文をすべて表示するには，単純に次のように入力すればよい．

```
scala> movies.text
res1: String =

          Pirates of the Caribbean
          Edward Scissorhands
```

上記の XML ファイル内の本文がすべて表示された．しかし，ブロック全体を一度に処理する必要はない．処理対象を絞り込むこともできる．Scala を用いて，XML 検索言語 XPath に似た照会言語を構築できる．ただし，//キーワードは Scala ではコメントなので，代わりに\と\\を使う．トップレベルのノードを検索するには，次のように 1 文字のバックスラッシュを使う．

```
scala> val movieNodes = movies \ "movie"
movieNodes: scala.xml.NodeSeq =
  <movie genre="action">Pirates of the Caribbean</movie>
  <movie genre="fairytale">Edward Scissorhands</movie>
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

この例では，XMLのmovie要素を検索している．個々のノードを検索するにはインデックスを指定する．

```
scala> movieNodes(0)
res3: scala.xml.Node = <movie genre="action">Pirates of the Caribbean</movie>
```

0番目の要素Pirates of the Caribbeanが見つかった．@記号を用いて個々のXMLノードの属性を検索することもできる．例えば，文書の最初の要素のgenre属性を検索するには，次のように指定する．

```
scala> movieNodes(0) \ "@genre"
res4: scala.xml.NodeSeq = action
```

この例では，Scalaでできる XML処理のほんの一部をお見せしただけだが，大体の感じは掴んでもらえたと思う．次に，簡単な文字列とのパターンマッチングについて見ていこう．

パターンマッチング

パターンマッチングを用いると，データの一部分に基づいてコードを条件分岐させることができる．

以下に，最も簡単な形式のパターンマッチングを示す．

▶ code/scala/chores.scala

```
def doChore(chore: String): String = chore match {
  case "clean dishes" => "scrub, dry"
  case "cook dinner" => "chop, sizzle"
  case _ => "whine, complain"
}
println(doChore("clean dishes"))
println(doChore("mow lawn"))
```

2つの家事，clean dishesとcook dinnerを定義する．各家事にはコードブロックが対応している．この例では，コードブロックは単に文字列を返すだけだ．最後に定義した家事は_，すなわちワイルドカードだ．Scalaは最初にマッチした家事に対応するコードブロックを実行する．いずれにもマッチしない場合は“whine, complain”を返す．

```
>> scala chores.scala
scrub, dry
whine, complain
```

ガード

パターンマッチングには装飾も可能だ．Prologではパターンマッチングに条件が関連付けられていることが多かった．Scalaで階乗計算を実装するには，各match文のガードに条件を指定する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/scala/factorial.scala

```
def factorial(n: Int): Int = n match {
    case 0 => 1
    case x if x > 0 => factorial(n - 1) * n
}

println(factorial(3))
println(factorial(0))
```

最初のパターンは 0 だが，2 番目のガードは case x if x > 0 という形式をとる．これは，x > 0 であるような任意の x にマッチする．同じ要領でさまざまな条件を指定できる．パターンマッチングで正規表現と型をマッチさせることもできる．後述の並行性のセクションでは，空クラスを定義し，それをメッセージとして使用する例を紹介する．

正規表現

Scala ではファーストクラスの正規表現が使える．.r メソッドを文字列に適用することで，任意の文字列を正規表現に変換できる．次ページに，文字列の先頭の大文字または小文字の F にマッチする正規表現の例を示す．

```
scala> val reg = """^(F|f)\w*""".r
reg: scala.util.matching.Regex = ^(F|f)\w*

scala> println(reg.findFirstIn("Fantastic"))
Some(Fantastic)

scala> println(reg.findFirstIn("not Fantastic"))
None
```

まず文字列を用意する．"""で囲むことで，複数行を許し，特殊文字の解釈を防ぐ．.r メソッドは文字列を正規表現に変換する．次に findFirstIn を用いて，最初にマッチする部分を検索する．

```
scala> val reg = "the".r
reg: scala.util.matching.Regex = the

scala> reg.findAllIn("the way the scissors trim the hair and the shrubs")
res9: scala.util.matching.Regex.MatchIterator = non-empty iterator
```

この例では，正規表現を構築し，findAllIn メソッドを用いて，文字列"the way the scissors trim the hair and the shrubs"内に登場するすべての the を検索する．必要なら，foreach を用いて，マッチ部分のリスト全体に対して繰り返すこともできる．正規表現については以上だ．文字列を使うときとまったく同じ要領で，正規表現とのマッチングを行うことができる．

XML とパターンマッチング

Scala で面白いのは，XML 構文とパターンマッチングの組み合わせだ．XML ファイルを読み込み，返されたさまざまな XML 要素に基づいてコードを条件分岐させる．例えば，次の XML 形式の映画タイトルファイルで考えてみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/scala/movies.scala

```
val movies = <movies>
    <movie>The Incredibles</movie>
    <movie>WALL E</movie>
    <short>Jack Jack Attack</short>
    <short>Geri's Game</short>
</movies>

(movies \ "_").foreach { movie =>
    movie match {
        case <movie>{movieName}</movie> => println(movieName)
        case <short>{shortName}</short> => println(shortName + " (short)")
    }
}
```

このコードはツリー内のすべてのノードについて，パターンマッチングを用いて，short（短篇映画）と movie（映画）とのマッチングを行う．筆者が気に入っているのは，Scala では，XML 構文，パターンマッチング，XQuery ライクな言語を用いて，最も一般的なタスクを拍子抜けするくらい簡単に片付けることができるところだ．結果はほとんど何の苦労もなく得られる．

以上，パターンマッチングの基本的な操作についてざっと見てきた．次の並行性のセクションでは，パターンマッチングを実際に使ってみる．

並行性

Scala の最も重要な側面は並行性の実現方法だ．並行性を実現している主な構文は，アクターとメッセージパッシングである．アクターは，スレッドプールとキューで構成される．アクターにメッセージを送信すると（!演算子を使用），オブジェクトがキューに登録される．アクターはメッセージを読み込んで，処理を実行する．多くの場合，アクターは，パターンマッチングを用いてメッセージを検出し，適切な処理を実行する．次の kids プログラムで考えてみよう．

▶ code/scala/kids.scala

```
import scala.actors._
import scala.actors.Actor._

case object Poke
case object Feed

class Kid() extends Actor {
  def act() {
    loop {
      react {
        case Poke => {
          println("Ow...")
          println("Quit it...")
        }
        case Feed => {
          println("Gurgle...")
          println("Burp...")
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
        }
      }
    }
  }
}

val bart = new Kid().start
val lisa = new Kid().start
println("Ready to poke and feed...")
bart ! Poke
lisa ! Poke
bart ! Feed
lisa ! Feed
```

このプログラムでは，2つの空の小さなシングルトン Poke と Feed を作成する．これらのシングルトンは何もしない．単に，メッセージとして使われるだけだ．このプログラムのキモは Kid クラスである．Kid はアクターだ．つまり，スレッドプールから起動され，キューからメッセージを取得する．アクターは1つメッセージを処理しては次のメッセージに進む．そこで，簡単な loop を開始する．ループ内にあるのは react 構文だ．react はアクターのメッセージを受信する．パターンマッチングによって，適切なメッセージ（この例では Poke または Feed のどちらか）とのマッチングを行うことができる．

スクリプトの残りの部分では，2人の子供（オブジェクト）を作成し，それらに Poke または Feed メッセージを送信して操作している．次のようにして実行する．

```
batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Ow...
Quit it...
Gurgle...
Burp...
Gurgle...
Burp...

batate$ scala code/scala/kids.scala
Ready to poke and feed...
Ow...
Quit it...
Gurgle...
Burp...
Ow...
Quit it...
Gurgle...
Burp...
```

実際に並行に実行されていることを示すため，アプリケーションを2回起動している．出力される応答の順序が違っている点に注目してほしい．アクターではタイムアウト応答を返すこともできる（reactWithin）．これは指定された時間内にメッセージを受信しないとタイムアウトを発生させるものだ．また，receive（スレッドをブロックする）や

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

receiveWithin（タイムアウト発生時にスレッドをブロックする）も使用できる．

並行処理の実例

これだけでは面白くないので，もう少し骨のあることをやってみよう．次のアプリケーション sizer では，Web ページのサイズを計算してみる．数ページ表示して，そのサイズを計算する．待ち時間がかなり長くなるので，アクターを用いてすべてのページを並行に取得するようにしたい．まずはプログラム全体を見てもらい，その後，個々のセクションについて見ていこう．

▶ code/scala/sizer.scala

```
import scala.io._
import scala.actors._
import Actor._

object PageLoader {
 def getPageSize(url : String) = Source.fromURL(url).mkString.length
}

val urls = List("http://www.amazon.com/",
                "http://www.twitter.com/",
                "http://www.google.com/",
                "http://www.cnn.com/" )

def timeMethod(method: () => Unit) = {
 val start = System.nanoTime
 method()
 val end = System.nanoTime
 println("Method took " + (end - start)/1000000000.0 + " seconds.")
}

def getPageSizeSequentially() = {
 for(url <- urls) {
   println("Size for " + url + ": " + PageLoader.getPageSize(url))
 }
}

def getPageSizeConcurrently() = {
 val caller = self

 for(url <- urls) {
   actor { caller ! (url, PageLoader.getPageSize(url)) }
 }

 for(i <- 1 to urls.size) {
   receive {
     case (url, size) =>
       println("Size for " + url + ": " + size)
   }
 }
}

println("Sequential run:")
timeMethod { getPageSizeSequentially }
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
println("Concurrent run")
timeMethod { getPageSizeConcurrently }
```

では先頭から解説していく．まず，並行性を実現し，HTTP リクエストを実行できるように，いくつか基本的なインポートを実行して actors および io 用のライブラリを読み込んでいる．次に，与えられた URL を用いてページサイズを計算する．

▶ code/scala/sizer.scala

```
object PageLoader {
 def getPageSize(url : String) = Source.fromURL(url).mkString.length
}
```

次に，いくつかの URL のリストを val に代入する．その後，各 Web リクエストの所要時間を計測するメソッドを作成する．

▶ code/scala/sizer.scala

```
def timeMethod(method: () => Unit) = {
 val start = System.nanoTime
 method()
 val end = System.nanoTime
 println("Method took " + (end - start)/1000000000.0 + " seconds.")
}
```

次に，2 つの異なる方法で Web リクエストを実行する．最初の方法は，forEach ループ内で各リクエストを回すシーケンシャルな処理方法だ．

▶ code/scala/sizer.scala

```
def getPageSizeSequentially() = {
 for(url <- urls) {
   println("Size for " + url + ": " + PageLoader.getPageSize(url))
 }
}
```

もう 1 つの方法では非同期に処理する．

▶ code/scala/sizer.scala

```
def getPageSizeConcurrently() = {
 val caller = self

 for(url <- urls) {
   actor { caller ! (url, PageLoader.getPageSize(url)) }
 }

 for(i <- 1 to urls.size) {
   receive {
     case (url, size) =>
       println("Size for " + url + ": " + size)
   }
 }
}
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

このアクターでは固定されたメッセージ・セットを受信することが分かっている．forEach ループ内では 4 つの非同期リクエストを送信する．これはほぼ一瞬で実行される．次に receive を用いて，単純に 4 つのメッセージを受信する．この部分がキモだ．最後にスクリプトを実行してテストする．

▶ code/scala/sizer.scala

```
println("Sequential run:")
timeMethod { getPageSizeSequentially }

println("Concurrent run")
timeMethod { getPageSizeConcurrently }
```

出力結果は次のようになる．

```
>> scala sizer.scala
Sequential run:
Size for http://www.amazon.com/: 81002
Size for http://www.twitter.com/: 43640
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Method took 6.707612 seconds.
Concurrent run
Size for http://www.google.com/: 8076
Size for http://www.cnn.com/: 100739
Size for http://www.amazon.com/: 84600
Size for http://www.twitter.com/: 44158
Method took 3.969936 seconds.
```

予想どおり並行ループのほうが速い．Scala の面白い問題の概要は以上だ．学んだことを次の演習で確認してみよう．

3 日目で学んだこと

3 日目は，量は少なかったがその分内容は濃かったと思う．異なる並行性プログラムを 2 つ作成し，アクター，パターンマッチング，正規表現を用いて，XML 文書を直接処理し，分散メッセージパッシングを行った．

この章全体を通して，互いに依存し合う 4 つの基本的な構文について学んだ．最初に，Scala で直接 XML を使う方法を学んだ．XQuery ライクな構文を用いて個々の要素や属性を問い合わせることができた．

次に Scala バージョンのパターンマッチングを紹介した．最初は簡単な case 文から始めたが，ガード，型，正規表現と紹介するうちに，その威力が明らかになった．

次に並行性の解説に移り，アクターを使った．アクターとは並行性を実現するために構築されたオブジェクトだ．通常は react または receive メソッドを loop でラップした形をしている．この receive メソッドが，キューに登録されたオブジェクトへのメッセージを受信する厄介な作業をすべてやってくれる．最後に内部パターンマッチングを行った．メッセージとしてはクラスをそのまま使った．これらのメッセージは，コンパクト，軽量，堅牢で，操作しやすいという特徴を備えている．メッセージ内にパラメータを指定する必要がある場合は，ページサイズを計算するアプリケーション内で URL に関

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

して行ったように，裸の属性をクラス定義に追加するだけでよい．

本書に登場するすべての言語と同じように，Scala はここで見たよりもはるかに堅牢だ．Java クラスとのやりとりは，この章でお見せしたものよりはるかに奥が深い．この章では，カリー化などの複雑な概念の上っ面を撫でたにすぎない．それでも，これから深く掘り下げて学習する向きには，良い基礎固めになったはずだ．

セルフスタディ 3 日目

Scala が提供している高度な機能をいくつか見てきたが，ここで実際に自分で試してみよう．いつものとおり 3 日目の演習は少し難しくなっている．

探してみよう：

- ページサイズ計算プログラムで，リンク先ごとに新しいアクターを作成しなかった場合，アプリケーションのパフォーマンスはどうなるだろうか．

試してみよう：

- ページサイズ計算アプリケーションに，ページ内のリンク数を数えるメッセージを追加してみよ．
- ボーナス問題：指定されたページ上のリンクをたどり，リンク先のページを読み込むようにしてみよ．例えば，“google.com” のサイズを計算するのであれば，Google とそのリンク先のすべてのページのサイズを計算する．

5.5　Scala のまとめ

Scala は，2 つのプログラミングパラダイムを強力にサポートしているため，これまでに紹介した他の言語よりも詳しく説明した．オブジェクト指向機能は，Java の代わりとして Scala の立場を確立するものである．Ruby や Io と違って，Scala は静的型付けを採用している．シンタックス面では，中括弧やコンストラクタの使い方など，Java から多くを流用している．

Scala は，関数型の概念と変更不能な変数も強力にサポートしている．また，並行性と XML を重視しているため，現在 Java で実装されている広範なエンタープライズアプリケーションにも適している．

Scala の関数型機能は，非常に充実しており，この章で説明しきれなかった部分も多数ある．例えば，カリー化，完全なクロージャ，複数のパラメータリスト，例外処理などの構文については説明しなかったが，これらはすべて，Scala のパワーと柔軟性を強化する価値のある概念だ．

Scala の長所と短所について見ていこう．

長所

Scala の強みの多くは，Java 環境と密接に統合された高度なプログラミングパラダイムといくつかのよく設計されたコア機能を中心とする．とりわけアクター，パターンマッ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:56:47 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

チング，XML 統合化は重要で，よく設計されている．一つひとつ見ていこう．

並行性

Scala による並行性の実現方法は，並行プログラミングの大きな進歩を象徴している．アクターモデルとスレッドプールは改良点として歓迎されており，可変状態なしでアプリケーションを設計する機能は並外れて充実している．

Io でも見たアクター・パラダイムだが，Scala のアクター・パラダイムは，開発者にとって理解しやすく，学界でも広く研究されている．Java と Ruby にはこの分野での改良を望みたいところだ．

並行性モデルは，並行性を実現する仕組みの一部でしかない．オブジェクトが状態を共有する場合，変更不能な値を奪い合うことになる．Io と Scala はこの点を少なくとも部分的には正しく認識し，可変状態を許可してはいるが，不変性をサポートするライブラリとキーワードも用意している．不変性は，コード設計を改良して並行性を実現するためにできる唯一の最も重要な点である．

最後に，Scala のメッセージパッシング構文は，次章の Erlang で見るメッセージパッシングとよく似ている．標準の Java スレッドライブラリと比べると，大幅に強化されている．

レガシー Java の進化

Scala は，強力な組み込みのユーザベース，つまり Java コミュニティから始まった．Scala のアプリケーションは，Java のライブラリを直接，または必要に応じてプロキシオブジェクトのコードを生成することによって間接的に利用できるため，互換性に優れている．型推論は，時代遅れの Java の型システムを改善する待望の改良点だ．新しいプログラミングコミュニティを確立するには，既存のコミュニティをすべて受け入れるのが一番だ．Scala は，Java のコンパクト化（これは価値のある考えだが）をうまく実現している．

Scala は，逆に Java コミュニティにも新しい機能をもたらした．コードブロックはファーストクラスの言語構文であり，基本的なコレクションライブラリとも相性が良い．また Scala により，トレイトという形でファーストクラスの mixin も提供された．パターンマッチングも大幅に強化された．こうした機能により Java の開発者は，高度な関数型パラダイムを特に意識することなく，高度なプログラミング言語を利用できる．

関数型構文を取り入れると，アプリケーションが大幅に改善される．Scala のアプリケーションは通常，同等の Java アプリケーションに比べて，コードの総行数が数分の 1 で済む．このことは非常に重要である．優れたプログラミング言語というものは，非常に複雑な概念を，オーバーヘッドを最小限に抑えて，より少ないコードで表現できなければならない．Scala はこの約束を果たしている．

ドメイン固有言語

Scala は，その柔軟なシンタックスと演算子オーバーロードによって，Ruby スタイルのドメイン固有言語を開発するのに理想的な言語となっている．Ruby 同様，演算子は単なるメソッドの宣言なので，ほとんどの場合オーバーライドできる．また，オプションの空白，ピリオド，セミコロンによって，多くの異なる形式をとることができる．これらは

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

堅牢な mixin とともに DSL の開発者が求めているツールである．

XML

Scala には XML サポートが組み込まれている．パターンマッチングにより，異なる XML 構文ブロックの解析が容易になる．複雑な XML 文書を深く探索できる XPath 構文が組み込まれているため，簡単で読みやすいコードを書くことができる．これは，とりわけ，XML 関連の処理が多い Java コミュニティにとって，歓迎すべき重要な改良点だ．

橋渡し

新しいプログラミングパラダイムが出現するたびに，橋渡し役の言語が必要になる．Scala はその橋渡し役としてふさわしい．関数型プログラミングモデルは重要である．というのは，このモデルでは並行性をうまく処理できるうえ，プロセッサ設計の進化に伴い並行性が組み込まれたプロセッサが増えているからだ．Scala を使うことにより，開発者は，関数型プログラミングに至る道を試行錯誤しながら進むことができる．

短所

筆者は Scala で採用されている多くの概念を気に入っているが，シンタックスに関しては覚えることが多く，学究的だと思っている．シンタックスは好みの問題とはいえ，Scala のシンタックスは他の大部分の言語に比べてプログラマの負担が大きい．特に老眼の身にはつらい．Scala を効果的な橋渡し役言語にしているいくつかの妥協点が，同時に Scala の価値を低めることになる点も認識している．ここでは Scala の弱点を 3 つだけ挙げるが，いずれも大きな弱点だ．

静的型付け

静的型付けは，関数型プログラミング言語には無理なく適合する型付けだが，Java スタイルのオブジェクト指向システム向け静的型付けは悪魔と関わるようなものだ．ときとして，開発者に負担を課して，コンパイラの要件を満たさなければならない．静的型付け言語における開発者の負担は，想像するよりもずっと大きい．コード，シンタックス，プログラム設計への影響は甚大だ．Scala を学習する過程で，筆者は言語のシンタックスといつも格闘し，プログラムの設計に苦労させられた．この負荷はトレイトによって若干軽減されてはいるが，プログラマの柔軟性とコンパイル時チェックの間のバランスは悪い．

純粋な関数型の強く静的な型システムがどのようなものかは，後で Haskell の章を読めば分かる．2 つのプログラミングパラダイムを同時に実現するという負担がなければ，Scala の型システムはずっと流動的かつ生産的なものになり，ポリモーフィズムのサポートも強化され，今と同等の利点を得るのにプログラマに求められる厳格さも少なくて済むだろう．

シンタックス

Scala のシンタックスは，見た目には少し学究的で難しく見える．シンタックスから受ける印象というのはいたって主観的なものなので，この項を短所に入れるかどうか迷ったのだが，一部，不可解な部分も見られるので入れることにした．Scala では，コン

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ストラクタなど，Java の規則をそのまま流用している部分もある．例えば `Person.new` とは書かず `new Person` と書くといった点だ．一方，新しい規則を導入している部分もある．引数の型などがそうだ．Java では `setName(String name)` と書くが，Scala では `setName(name: String)` と書く．戻り型は，メソッド宣言の先頭ではなく末尾に指定する．こうしたちょっとした違いがあるため，シンタックスに気をとられコードに集中できない．問題は，Scala と Java の間で行き来する際に，本来必要ないはずの労力を強いられる点だ．

可変性

橋渡し言語を構築するときは，妥協点を考慮に入れる必要がある．Scala の重要な妥協点のひとつに，可変性の導入がある．Scala では，`var` によって，ある意味パンドラの箱を開けてしまった．というのは，可変状態を許すと，並行処理時にさまざまなバグが発生する可能性があるからだ．しかし，丘の上の城に住んでいた特別な青年を家に連れてくるのであれば，こうした妥協もやむを得ないところだ．

終わりに

全般に，筆者の Scala 体験はさまざまな感情が入り交じったものとなった．静的型付けには驚いた．と同時に，Java 開発者でもある筆者は，強化された並行性モデル，型推論，XML 処理を大いに評価している．Scala は最先端のプログラミング技術を大きく前進させた．

もし筆者が，Java のプログラムやプログラマにかなりの投資をしているなら，Scala を用いて生産性の向上を図るだろう．また，かなり大規模なスケーラビリティと並行処理を必要とするアプリケーションを開発するなら，Scala を検討するだろう．商業的には，このフランケンシュタイン言語は，オブジェクト指向言語と関数型言語の橋渡し役として，また Java の大規模なプログラミングコミュニティを完全に受け入れることによって，かなり成功している．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第 6 章

Erlang

聞こえるか，アンダーソン君．逃れられない運命が近づいてくるのが．
エージェント・スミス

Erlang のような謎めいた言語はほとんど見かけない．Erlang は，難しいことを簡単にし，簡単なことを難しくする並行処理指向言語だ．その仮想マシン *BEAM* は，堅牢なエンタープライズ向けとして優れており，これと同等の機能を持つ仮想マシンは Java 仮想マシンくらいである．効率は高く，怪我さえしかねないほどだが，Ruby のような美しく単純なシンタックスは備えていない．映画『マトリックス』のエージェント・スミスを思い浮かべてほしい[*1]．

『マトリックス』は 1999 年に公開された SF 映画の傑作だ．この映画では，現実世界が，実はコンピュータが作り保守している仮想世界，すなわち幻想として描かれている．エージェント・スミスは，マトリックス内の人工知能プログラムであり，姿形を自由に変えることができ，複数の場所に同時に存在できるという，現実にはあり得ない驚くべき能力を備えている．彼から逃れることは不可能だ．

6.1 Erlang について

Erlang というのは奇妙な名前だが，Ericsson Language の略であり，デンマークの数学者 Agner Karup Erlang の名前でもあるこの名称はなぜかぴったりくる[*2]．Erlang 氏は電話通信網の解析に用いる数学の権威である．

1986 年，Joe Armstrong が Ericsson で最初のバージョンを開発し，80 年代の後半の 5 年間を通して開発と改良を続けた．1990 年代には断続的に成長を遂げ，2000 年代にはさらに支持を獲得した．人気のあるクラウドコンピューティング向けデータベース CouchDB と SimpleDB は，Erlang で書かれている．Erlang は Facebook のチャット機能にも使われている．Erlang 人気は確実に広がっている．というのは，Erlang は他の多くの言語にはない特徴を備えているからだ．それはスケーラブルな並行性（concurrency：コンカレンシ）と信頼性だ．

*1 『マトリックス』. DVD. Andy Wachowski，Lana Wachowski 監督作品．1999 年．カリフォルニア州バーバンク，Warner Home Video，2007 年．

*2 ［訳注］Erlang（プログラミング言語）の発音は，日本語で表記すると「アーラン」が近いようだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

言語レベルの並行性

Ericsson 社は，数年間にわたって，準リアルタイムで動作し耐障害性のある通信事業者向けの分散アプリケーションを開発するために研究を重ねていた．その成果が Erlang である．通信事業者向けのシステムは保守時にもダウンさせることができず，ソフトウェア開発にも法外な費用がかかった．そこで Ericsson 社は，1980 年代を通してプログラミング言語について調査し，さまざまな理由で，既存の言語では同社のニーズを満たすには不十分であると悟った．結局，こうした要件を満たすためにまったく新しい言語が開発されることになる．

Erlang は，多数の信頼性機能が組み込まれた関数型言語だ．Erlang を使えば，信じられないくらい信頼性の高いシステムを実現できる．電話交換システムは保守作業中でもダウンさせられないが，Erlang ならダウンさせなくてもモジュールをまるごと入れ替えることができる．Erlang を使ったアプリケーションのなかには，保守中でもダウンすることなく数年にわたって稼働し続けているものもある．こうした信頼性もさることながら，Erlang の最大の特徴はその並行性である．

どのアプローチが最善かについては，並行処理の専門家の間でも意見が分かれることがある．よくあるのは，スレッドとプロセスのどちらが並行性を高めるかという議論だ．多数のスレッドが 1 つのプロセスを形成する．プロセスは固有のリソースを持っている．スレッドはそれぞれ固有の実行パスを有するが，同じプロセス内の他のスレッドとリソースを共有する．通常，スレッドのほうがプロセスより軽量であるが，実装はさまざまだ．

スレッドは使わない

Java や C など，多くの言語がスレッド方式で並行処理を実現している．スレッドはリソースの消費量が少ないため，理論上はスレッドのほうが高いパフォーマンスを実現できるはずである．スレッドの欠点として，リソースを共有するため実装が複雑になりバグが発生しやすいこと，リソースをロックすることになるため，それがボトルネックになることが挙げられる．スレッドを用いて，リソースを共有する 2 つのアプリケーション間の協調制御を行うには，セマフォまたはオペレーティングシステムレベルのロックが必要になる．Erlang はこれとは異なるアプローチをとる．すなわち，プロセスをできる限り軽量にするというやり方だ．

軽量プロセス

Erlang では，共有リソースとリソースボトルネックの泥沼にはまるのを避け，軽量プロセスという考え方を採用している．Erlang の開発者は，多数のプロセスを使ったアプリケーション内でのプロセスの作成と管理，およびプロセス間通信を簡素化しようとさまざまな努力を行った．分散メッセージパッシングは基本的な言語レベルの構造であり，ロックの必要性を排除し，並行性を高める．

Io と同様に Erlang でもアクターを用いて並行処理を実現しているため，メッセージパッシングは重要な概念だ．Scala にもメッセージパッシング構文があるが，Erlang のメッセージパッシングもこれと似ている．Scala ではアクターはオブジェクトで表現され，スレッドプールが動作基盤となっている．Erlang ではアクターは軽量プロセスである．アクターはキューから着信メッセージを読み出し，パターンマッチングを使ってメッ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

セージの処理方法を決定する．

信頼性

Erlang も通常のエラーチェックの仕組みを備えてはいる．しかし通常の耐障害性アプリケーションに比べてエラー処理の回数ははるかに少ない．Erlang のモットーは「クラッシュさせろ」だ．Erlang ではプロセスの死を監視するのは簡単なので，関連するプロセスを強制終了して新しいプロセスを開始しても大した負荷にならない．

コードのホットスワップ（つまり，コードを停止せずにアプリケーションの一部分を入れ替えること）も可能だ．この機能を使うと，同じような分散アプリケーションよりもはるかに保守が簡単になる．Erlang では，堅牢な「クラッシュさせろ」式のエラー対処方法に，ホットスワップと（最小限のオーバーヘッドで起動できる）軽量プロセスを組み合わせている．複数のアプリケーションが同時に何年にもわたってダウンタイムなしで稼働しているといったケースはいくらでもある．

Erlang の並行処理機能はとても魅力的だ．並行処理に欠かせない基本要素，すなわちメッセージパッシング，プロセスの生成，プロセスの監視がすべて揃っている．生成されるプロセスは軽量なので，よくあるリソースの制約について心配する必要はない．副作用と可変性を極力排除するように設計されており，プロセスの死を監視するのは簡単だ（些細なことといってもよいくらいだ）．こうした特徴の組み合わせにより，Erlang の並行処理機能は誰でも使ってみたくなる魅力を備えている．

Joe Armstrong 博士へのインタビュー

本書の執筆を通して，筆者が最も尊敬する何人かの人物とお会いする（あるいはメールをやりとりする）機会があった．Erlang の開発者であり，『プログラミング Erlang』[Arm07] の著者でもある Joe Armstrong 博士（ストックホルム出身）は，なかでも最もお会いしたかった人物だ．Erlang の最初の実装者である博士についにお話を伺うことができた．

Bruce　Erlang を開発した理由は何ですか．

Dr. Armstrong　偶然です．新しいプログラミング言語を開発しようと思っていたわけではありません．我々は当時，電話交換制御用ソフトウェアを書くために，何か良い方法を模索していました．それで，Prolog をいじり始めたのです．Prolog は素晴らしい言語でしたが，私が求めていた言語とは少し違っていました．そこで Prolog にいろいろと手を加え始めました．「Prolog の挙動を変えてみたらどうなるだろう」と考え，Prolog のメタインタープリタを書いたのです．具体的には，Prolog に並列プロセス機能，エラー処理の仕組みなどを追加しました．その後しばらくして，これらの一連の変更が加えられた Prolog に Erlang という名前が付けられ，新しい言語が誕生しました．プロジェクトへの参加人数が増えるにつれ，Erlang は成長していきました．コンパイルもできるようになり，機能も追加され，ユーザも増えました．

Bruce　Erlang のどのような点が最も気に入っていますか？

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Dr. Armstrong　エラー処理機構，動的なコード更新メカニズム，ビットレベルのパターンマッチングといったところでしょうか．エラー処理機構は，Erlang の中でも最も理解されていない部分であり，他の言語との違いが最も大きい部分でもあります．Erlang プログラミングのモットーである「非防御的」プログラミングと「クラッシュさせろ」という概念はすべて，従来の慣習と正反対ですが，そのおかげで本当にコンパクトで美しいプログラムが書けるのです．

Bruce　一から作り直せるとしたら，どの部分を変更しますか？（Erlang の最も大きな制限は何ですか？）

Dr. Armstrong　それは難しい質問です．おそらく日によって答えが違ってくるのではないかと思います．モビリティを強化して，実行中の処理をインターネットを通じて渡せるようになるといいですね．これはライブラリレベルではできますが，言語ではサポートされていません．今すぐと言われれば，Erlang のルーツに戻って Prolog ライクな述語論理を追加できたら素晴らしいと思います．述語論理とメッセージパッシングの新たな融合です．
ハッシュマップや高階モジュールなど，他にもあればいいなと思う小さな変更点はたくさんあります．
一からやり直すとしたら，部品を組み合わせる方法にもっと気を配るでしょうね．例えば，大量のコードで構成される大規模プロジェクトの実行方法（コードのバージョン管理，コードの検索方法，コードの進化の管理）などです．コードの量が膨大になると，プログラマの仕事も，一からコードを書く作業から，既存のコードを検索して統合する作業へと変わってきます．ですから，コードを検索し，必要な部品を組み合わせる機能がますます重要になってきます．Git や Mercurial から得たアイデアや，型システムなどを言語自体に組み込んで，コードが進化する様子を管理された方法で把握できるようになれば素晴らしいでしょうね．

Bruce　Erlang が実際に使われている環境をいろいろと見てきて，最も驚いたのはどんな環境ですか．

Dr. Armstrong　驚いたことはないですね．そうした使われ方をするのは分かっていましたから．私は，Ubuntu を Karmic Koala（Ubuntu 9.10）にアップデートしたのですが，アップデート後，見事に隠ぺいされた Erlang がバックグラウンドで動作していることに気づきました．これは，CouchDB をサポートするためです（CouchDB も私のマシン上で稼働していました）．この自動インストールの仕組みは，Erlang にいわば仕込まれ，誰にも気づかれないで 1,000 万台のマシンに導入されたのです．

この章では，Erlang の基礎について説明した後，関数型言語としての Erlang を実際に試してみる．さらに，並行性といくつかのカッコいい信頼性機能についても，少しページを割いて説明する．そう，カッコいい信頼性機能というのもあり得るのだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

6.2　1 日目：外見は普通

エージェント・スミスは，マトリックスという仮想現実世界を破壊しようとするプログラム（シミュレートされた人間）を抹殺するプログラムである．スミスを危険ならしめている最も基本的な特性は，彼が自身を人間のように見せることができる能力だ．1 日目は，Erlang でごく普通のアプリケーションも書けるという点を見ていこう．解説ではできる限り「普通」に見えるように最善を尽くすつもりだが，Erlang ではそれは簡単なことではない．

純粋なオブジェクト指向プログラマとして本書を読み始めた方は，ここまで読み進めてきて少し悪戦苦闘しているかもしれないが，そんなに戦闘モードになることはない．コードブロックについては Ruby の章で，アクターについては Io の章で，パターンマッチングについては Prolog の章で，分散メッセージパッシングについては Scala の章で，既に説明済みだ．これらは Erlang の基本原理となる．この章では，別の重要な概念を導入する．Erlang は本書で最初に登場する関数型言語だ（Scala は関数型とオブジェクト指向型のハイブリッド言語だ）．これは次のことを意味する．

- プログラムはすべて関数で作成する．オブジェクトはどこにも登場しない．
- 関数は通常，与えられた入力が同じであれば，同じ値を返す．
- これらの関数は通常，副作用を持たない（プログラムの状態を変更しない）．
- すべての変数への代入は 1 回だけに限られる．

1 番目のルールに従うのは難しそうだがまったく無理というわけではない．残りの 3 つのルールに従うのは少なくともしばらくは無理かもしれない．これらのルールに従ってコーディングする方法を学べば，徹底した並行処理向けのプログラムが出来上がる．可変状態という問題が排除されれば，並行処理は驚くほど簡単になる．

注意深く読んでみると，2 番目と 3 番目のルールで通常という言葉が使われているのが気にかかる．Erlang は純粋な関数型言語ではない．いくつか例外を認めているからだ．本書で紹介する言語のうち，純粋な関数型言語は Haskell だけである．とはいうものの，Erlang でも関数型プログラミングの特徴は強く感じることができるし，たいていは上記のルールに従ってコーディングする．

はじめよう

筆者は Erlang バージョン R13B02 を使っているが，本章で紹介する基本的な機能はどのバージョンでも問題なく動作するはずだ．Erlang のシェルを起動するには，コマンド行に `erl`（一部の Windows システムでは `werl`）と入力する．

```
batate$ erl
Erlang (BEAM) emulator version 5.4.13 [source]

Eshell V5.4.13  (abort with ^G)
1>
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

他の章でもそうしたように，ほとんどの作業は最初，コマンド行で行う．Java と同様に Erlang はコンパイル言語である．ファイルをコンパイルするには c(filename)．コマンドを用いる．末尾のピリオドは必須だ．ループから抜けるには Control+C を押せばよい．では始めよう．

コメント，変数，式

まずは，いくつか基本的なシンタックスを紹介しておく．コンソールを起動して，次のように入力してほしい．

```
1> % This is a comment
```

このようにコメント行は % で始める．行末まですべてがコメントとなる．1 つのコメントは，単一のスペースとして解析される．

```
1> 2 + 2.
4
2> 2 + 2.0.
4.0
3> "string".
"string"
```

各文はピリオドで終了する．基本的な型として文字列，整数，浮動小数点数がある．リストは次のように指定する．

```
4> [1, 2, 3].
[1,2,3]
```

Prolog 系の言語がそうであるように，リストは角括弧で囲む．次の例は少し驚くかもしれない．

```
4> [72, 97, 32, 72, 97, 32, 72, 97].
"Ha Ha Ha"
```

つまり，String は実際には List なのだ．そしてエージェント・スミスがあなたの母親を見て笑ったというわけだ．まったく社交性のかけらもない奴だ．2 + 2.0 の結果から分かるように，いくつかの基本的な型変換は行ってくれる．型エラーを発生させてみよう．

```
5> 4 + "string".
** exception error: bad argument in an arithmetic expression
 in operator  +/2
    called as 4 + "string"
```

Scala と違って，文字列と整数の間で型変換は行われない．変数への代入はどうだろう．

```
6> variable = 4.
** exception error: no match of right hand side value 4
```

これは，エージェントと Erlang に共通する嫌な部分の一例だ．この厄介な言語はときどき，情け容赦なく冷酷になる．このエラーメッセージは Erlang のパターンマッチングに関するものだ．エラーになったのは variable がアトムだからだ．変数は大文字で始

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 10:59:03 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

める必要がある．

```
7> Var = 1.
1
8> Var = 2.

=ERROR REPORT==== 8-Jan-2010::11:47:46 ===
Error in process <0.39.0> with exit value: {{badmatch,2},[{erl_eval,expr,3}]}

** exited: {{badmatch,2},[{erl_eval,expr,3}]} **
8> Var.
1
```

見てのとおり，変数は大文字で始まり，変更不能である．値は1度しか代入できない．この概念に，関数型言語に初めて出会ったプログラマの大半はとまどう．次に，もう少し複雑なデータ型を紹介しよう．

アトム，リスト，タプル

関数型言語では，シンボルの重要性が高い．シンボルは最も基本的なデータ要素であり，名前を付けたいものであれば何でも表すことができる．本書に登場した他の言語でもシンボルが出てきた．Erlangでは，シンボルはアトムと呼ばれ，小文字で始める．アトムは，何かを表すときに使う，最小単位となる値だ．次のように使う．

```
9> red.
red
10> Pill = blue.
blue
11> Pill.
blue
```

redとblueはアトム，すなわち実世界のものを記号化して扱うために使える任意の名前だ．最初の例ではredという簡単なアトムを返している．次の例ではblueというアトムをPillという変数に代入している．アトムは，より堅牢なデータ構造体に結び付けて使うともっと面白くなる．これについては後述する．ここではリストを眺めながら応用的な操作について見ていこう．リストは角括弧で囲んで表す．

```
13> [1, 2, 3].
[1,2,3]
14> [1, 2, "three"].
[1,2,"three"]
15> List = [1, 2, 3].
[1,2,3]
```

リスト構文はおなじみのものだ．リストには異なる型を混在させることができる．リストの長さに制限はない．リストは，基本データ型と同じように変数に代入できる．タプルは固定長の異種型混在リストだ．

```
18> {one, two, three}.
{one,two,three}
19> Origin = {0, 0}.
{0,0}
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ここは驚くようなことは何もない．Prolog の影響を強く受けている部分だ．後でパターンマッチングについて説明するとき気づくと思うが，タプルがマッチするにはサイズが重要になる．3 要素のタプルと 2 要素のタプルはマッチしない．リストをマッチさせるときは，長さが一致しなくてもよい．これは Prolog と同じだ．

Ruby ではハッシュマップを用いて名前と値を対応付ける．Erlang では，マップやハッシュを使う箇所でもタプルを使うことが多い．

```
20> {name, "Spaceman Spiff"}.
{name,"Spaceman Spiff"}
21> {comic_strip, {name, "Calvin and Hobbes"}, {character, "Spaceman Spiff"}}.
{comic_strip,{name,"Calvin and Hobbes"},
             {character,"Spaceman Spiff"}}
```

上の例は，新聞の連載マンガをハッシュで表したものだ．ハッシュキーにアトムを，ハッシュ値に文字列を使っている．リストとタプルを混在させることもできる．例えば，タプルで表されたマンガのリストを作成できる．個々の要素にはどのようにしてアクセスするのだろうか？ Prolog をまだ覚えている方は予想がつくだろう．そう，パターンマッチングを使う．

パターンマッチング

Prolog の章を読んだ方は，パターンマッチングの基本事項についてはしっかりと身についているはずだ．Prolog の場合との大きな相違点について指摘しておきたい．Prolog では，プログラマがルールを定義することによって，データベース内のすべての値とマッチングを行い，Prolog にすべての組み合わせを見つけだしてもらう．Erlang は Scala のように動作する．マッチングは単一の値に対して作用する．では，パターンマッチングを用いてタプルから値を取り出してみよう．ある人物を表すタプルを次のように定義する．

```
24> Person = {person, {name, "Agent Smith"}, {profession, "Killing programs"}}.
{person,{name,"Agent Smith"},
        {profession,"Killing programs"}}
```

名前を Name に，職業を Profession に代入したい（取り出したい）とする．次のマッチングによって値を取り出すことができる．

```
25> {person, {name, Name}, {profession, Profession}} = Person.
{person,{name,"Agent Smith"},
        {profession,"Killing programs"}}
26> Name.
"Agent Smith"
27> Profession.
"Killing programs"
```

Erlang は，データ構造体をマッチングすることで，変数をタプル内の各値に代入する．アトムは自分自身にマッチするため，結局このマッチングでは，変数 Name に"Agent Smith"が，変数 Profession に"Killing programs"がマッチする（代入される）ことになる．この機能は Prolog の場合とほぼ同じように動作し，基本的な意志決定（条件分岐）構文となる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Rubyや Javaのハッシュに慣れている方には，先頭に person というアトムがきている点が奇妙に思えるかもしれない．Erlangでは，複数のマッチング文と複数種類のタプルを使うことがよくある．このようにデータ構造体を設計しておけば，他のタプルを除外して，すべての person タプルだけを迅速にマッチングできる．

リストのパターンマッチングは Prologの場合と同じだ．

```
28> [Head | Tail] = [1, 2, 3].
[1,2,3]
29> Head.
1
30> Tail.
[2,3]
```

簡単だ．リストのヘッドで複数の変数をバインドすることもできる．

```
32> [One, Two|Rest] = [1, 2, 3].
[1,2,3]
33> One.
1
34> Two.
2
35> Rest.
[3]
```

リスト内の要素数が不足している場合，パターンは一致しない．

```
36> [X|Rest] = [].
** exception error: no match of right hand side value []
```

これで，他のエラーメッセージも少しは理解できるだろう．例えば，変数を大文字で始めるのを忘れたとしよう．次のようなエラーメッセージが表示される．

```
31> one = 1.
** exception error: no match of right hand side value 1
```

前述のとおり，=は単純な代入ではなく，実はパターンマッチングだ．上の例は，整数の 1をアトムの one と一致させるように Erlangに指示しているのだが，それはできないのでエラーになっている．

ビット単位のマッチング

データにビット単位でアクセスしなければならない場合がある．狭い領域に多くのデータを詰め込む場合，あるいは JPEGや MPEGなどのフォーマットを扱う場合には，各ビットの位置が重要になる．Erlangでは，複数のデータを 1バイトに簡単にパックすることができる．これらを実行するには 2つの操作，すなわちパックとアンパックが必要になる．Erlangでは，ビットマップは他のタイプのコレクションとまったく同じように動作する．データ構造体をパックするには，次のように各要素に必要なビット数を指示してやるだけでよい．

```
1> W = 1.
1
2> X = 2.
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
2
3>
3> Y = 3.
3
4> Z = 4.
4
5> All = <<W:3, X:3, Y:5, Z:5>>.
<<"(d">>
```

このコンストラクタでは，<<と>>でバイナリパターンを囲む．この例は，変数 W に 3 ビット，X に 3 ビット，Y に 5 ビット，Z に 5 ビットを使うという意味になる．次に，アンパックできなければならない．構文は想像どおりだ．

```
6> <<A:3, B:3, C:5, D:5>> = All.
<<"(d">>
7> A
7> .
1
8> D.
4
```

タプルやリストと同様，同じ構文（ビットパターン）を左辺に書いておけば，あとはパターンマッチングがアンパックしてくれる．このビット単位の操作により，Erlang のローレベルの処理機能は驚くほど強力になっている．

ここからは，多くの基本事項をざっと説明していく．というのは，本章に登場する主要なすべての概念は既に紹介済みだからだ．1 日目はこれでほとんど終わりなのだが，1 日目を終える前に，最も重要な概念である関数について説明しておく必要がある．

関数

Scala と違って，Erlang は動的に型付けされた言語だ．したがって，データ要素に型を割り当てることを心配する必要はない．Ruby と同様，Erlang の型付けは動的に行われる．Erlang は，引用符や小数点などのシンタックス上のヒントに基づいて，実行時に型のバインドを行う．ここでコンソールを再起動することにする．いくつか用語を紹介しておこう．関数は.erl という拡張子を持つファイルに格納する．このファイルにはモジュールのコードが格納される．このファイルを実行するにはコンパイルする必要がある．ファイルをコンパイルすると.beam という実行可能ファイルが生成される．.beam コンパイル済みモジュールは beam という仮想マシン内で動作する．

前置きはこのくらいにして，いくつか関数を作成してみよう．

次のようなファイルを入力する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/erlang/basic.erl

```
-module(basic).
-export([mirror/1]).

mirror(Anything) -> Anything.
```

1行目はモジュール名を定義している．2行目ではモジュールの外で使う関数を定義（エクスポート）している．この関数はmirrorという名前で，/1は1つの引数をとるという意味だ．最後に関数自体を定義している．Prolog形式のルールの影響を受けているのが分かる．関数の定義では，関数に名前を付け，引数を指定する．その後のシンボル->は，単純に第1引数を返しているだけだ．

関数の定義を終えたら，このコードが置かれているディレクトリからコンソールを起動し，次のようにしてコンパイルする．

```
4> c(basic).
{ok,basic}
```

basic.erlをコンパイルした．これで，コードファイルと同じディレクトリにbasic.beamというファイルが作成される．次のようにして実行する．

```
5> mirror(smiling_mug).
** exception error: undefined shell command mirror/1
6> basic:mirror(smiling_mug).
smiling_mug
7> basic:mirror(1).
1
```

このように関数名だけを指定しても実行できない．モジュール名とコロンを関数名の前に付ける必要がある．この関数は引数を出力するだけの簡単なものだ．

Anythingを2つの異なる型にバインドできている点に注意してほしい．Erlangでは動的に型付けされる．これは，少なくとも筆者にとっては快適だ．Scalaの強い型付けのあとだけに，週末を極寒のシベリアで過ごした後に帰宅したような気分だ．

もう少し複雑な関数を見てみよう．次の例では複数のマッチング・ケースを定義している．

matching_function.erlというファイルを作成する．内容は次のとおりだ．

▶ code/erlang/matching_function.erl

```
-module(matching_function).
-export([number/1]).

number(one)   -> 1;
number(two)   -> 2;
number(three) -> 3.
```

次のようにして実行する．

```
8> c(matching_function).
{ok,matching_function}
9> matching_function:number(one).
1
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
10> matching_function:number(two).
2
11> matching_function:number(three).
3
12> matching_function:number(four).
** exception error: no function clause matching matching_function:number(four)
```

紹介してきた関数のなかで，これは複数のマッチングの可能性がある最初の関数だ．各マッチには，関数名，マッチの対象となる引数，そして->シンボルの後に実行するコードを指定する．各ケースにマッチしたら，整数を返しているだけだ．最後の文は．で，他の文は；で終端する．

Io，Scala，および Prolog と同様，Erlang でも再帰は重要な役割を果たす．Prolog と同じように，Erlang では末尾再帰が最適化される．以下にお決まりの階乗計算のコードを示す．

▶ code/erlang/yet_again.erl

```
-module(yet_again).
-export([another_factorial/1]).
-export([another_fib/1]).

another_factorial(0) -> 1;
another_factorial(N) -> N * another_factorial(N-1).

another_fib(0) -> 1;
another_fib(1) -> 1;
another_fib(N) -> another_fib(N-1) + another_fib(N-2).
```

Erlang で階乗計算を行うとこうなる．他の言語と同じように再帰的に定義されている．ついでに，フィボナッチ数列を生成するコードも含めておく[*3]．

では実行してみよう．今度は時間をかけてコーディングした甲斐のある結果をお見せできると思う．

```
18> c(yet_again).
{ok,yet_again}
19> yet_again:another_factorial(3).
6
20> yet_again:another_factorial(20).
2432902008176640000
21> yet_again:another_factorial(200).
78865786736479050355236321393218506229513597768717326329474253324435944996340334292030428401198462390417721213891963883025764279024263710506192662495282993111346285727076331723739698894392244562145166424025403329186413122742829485327752424240757390324032125740557956866022603190417032406235170085879617892222278962370389737472000000000000000000000000000000000000000000000000
22> yet_again:another_factorial(2000).
33162750924506332411753933805763240382811720810578039457193543706038077905600822400273230859732592255402352941225834109258084817415293796131386633526343688905634058556163940605117252571870647856393544045405243
```

[*3] ［訳注］ここでは fib(0) が 1 を返すように定義されているが，fib(0) は 0 とするのが一般的だろう（another_fib(0) -> 0;）．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
95746703767410872297043468415834375243158087753364512748799543685924 74
... and on and on...
000000000000000000000000000000000000000000000000000000000000000000000000
```

素晴らしい．確かに今までとは違う．エージェント・スミスと Erlang に共通するすごい一面がようやく顔をのぞかせてきた．実行してみなかった人のために言っておくが，結果は本当に一瞬で表示される．整数の最大サイズがいくつだか知らないが，これは相当に大きい数だ．

スタートとしては悪くない．いくつか簡単な関数を作成して，実際に動かしてみた．では，そろそろ 1 日目のまとめに入ることにしよう．

1 日目で学んだこと

Erlang は関数型言語で，強く動的な型付けを採用している．シンタックスはシンプルだが，典型的なオブジェクト指向言語とはまったく異なる．

Prolog と同様，Erlang にもオブジェクトという概念はない．Erlang と Prolog は非常に密接な関連がある．パターンマッチング構文や複数の関数エントリポイントなどはおなじみのはずだし，一部の問題は，再帰を用いて同じ方法で解くことができる．関数型言語には，可変状態，さらには副作用という概念さえない．このためプログラムの状態を保持するのはぎこちなくなるが，これから新しい方法をいろいろと学ぶ．また，良い面も見えてくる．状態と副作用を排除することで，並行性の管理方法が驚くほど楽になる．

1 日目は，コンソール上で作業し，コンパイラも利用した．主に基本事項に的を絞って説明した．基本的な式を作成し，簡単な関数も作成した．Prolog と同様に Erlang でも，関数に複数のエントリポイントを持たせることができる．基本的なパターンマッチングも使った．

基本的なタプルとリストも扱った．タプルは Ruby のハッシュに相当し，データ構造体の基盤を形成する．リストとタプルによるパターンマッチングについて学んだ．これらの概念を理解することにより，後の章で，振る舞いをタプルやプロセス間メッセージに容易に関連付けることができる．

2 日目では，基本的な関数の概念を拡張する．コンカレントな世界で動作するコードを構築する方法について学ぶが，実際にはそこまではまだ行かない．少し時間を割いて演習問題を解き，ここまでに学んだことを実際に使ってみよう．

セルフスタディ 1 日目

Erlang のオンラインコミュニティは急速に拡大している．サンフランシスコでのカンファレンスもどんどん規模が大きくなっている．また Io や C と違って，単に Erlang と入力するだけで Google で必要な情報を見つけられるはずだ．

探してみよう：

- Erlang のオフィシャルサイト
- Erlang の関数ライブラリの正式ドキュメント
- Erlang の OTP ライブラリのドキュメント

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

試してみよう：

- 再帰を用いて文字列を構成する単語の数を返す関数を書け．
- 再帰を用いて 10 まで数える関数を書け．
- {error, Message}または success という形式の入力が与えられたとき，マッチングを用いて，“success” または “error: message” のどちらかを出力する関数を書け．

6.3　2 日目：変身

2 日目は，エージェント・スミスのパワーを堪能してもらおう．『マトリックス』に登場するエージェントは，超人的な力を備えている．弾丸を避けたり，コンクリートを拳で打ち抜いたりできる．関数型言語は，オブジェクト指向言語よりも抽象化のレベルが高い．このため理解は難しいものの，少ないコードで複雑な概念を表現できる．

エージェント・スミスは，マトリックス内の他のどの人間にでも姿を変えることができる．これは，関数型言語でも重要な能力だ．関数をリストに適用して，望みの形に手早く変換する方法をこれから学習する．例えば買い物リストを価格リストに変換したり，URL のリストをコンテンツと URL が格納されたタプルに変換したりできる．こうした問題は，関数型言語を使えばあっという間に解決できる．

制御構文

まずは Erlang で必須の基本制御構文について説明しよう．このセクションは Scala の場合より随分と短いことに気づくだろう．Erlang のプログラムでは，並行アプリケーションを書くときに処理すべきメッセージを解釈するため，case 文を使うことが多い．if 文は主流ではない．

case

では case 文から始めよう．ほとんどの場合，パターンマッチングといえば関数呼び出しのコンテキストを思い浮かべるのだが，この構文はどこでも使えるパターンマッチングと考えればよい．例えば Animal という変数の値に基づいてコードを条件分岐させたいとする．

```
1> Animal = "dog".
2> case Animal of
2>     "dog" -> underdog;
2>     "cat" -> thundercat
2> end.
underdog
```

この例では，文字列"dog"が最初の節にマッチするので，underdog というアトムが返される．Prolog と同様，任意の値にマッチさせるにはアンダースコア（_）を使う．次に例を示す（Animal は"dog"のままで変わらない）．

```
3> case Animal of
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
3>     "elephant" -> dumbo;
3>     _ -> something_else
3> end.
something_else
```

Animal は"elephant"ではないので，最後の節にマッチする．アンダースコアは Erlang の他のマッチングでも使える．ここで基本的なシンタックス上の欠点を指摘しておきたい．すべての case 節はセミコロンで終わるが，最後の節だけは例外だ．このため，文を編集して節の順序を変更する場合は，末尾のセミコロンも追加削除する必要がある．最後の節にもセミコロンを書くことが許されれば編集作業が楽になっただろう．もちろん，このシンタックスは理にかなっている．セミコロンは case 節の区切り文字なのだ．ただ，少し使いにくい．エージェント・スミスが，私の小さな甥っ子に笑いながら足で砂をかけたら，こんな気分になるだろうか．「今月の優秀エージェント」賞を獲得したければ，彼はもう少し対人関係のスキルを磨く必要がある．では次に基本的な if 文の使い方を説明しよう．

if

case 文ではパターンマッチングを使うが，if 文ではガードを使う．Erlang のガードは，マッチが成功するために満たす必要がある条件のことだ．パターンマッチングでガードを使う方法を後で紹介するが，ガードの最も基本的な形は if 文に現れる．最初に if キーワードを指定し，その後に複数のガード -> 式節が続く．次のような感じだ．

```
if
  ProgramsTerminated > 0 ->
    success;
  ProgramsTerminated < 0 ->
    error
end.
```

どの節にもマッチしなかったらどうなるのだろうか？

```
8> X = 0.
0
9> if
9>   X > 0 -> positive;
9>   X < 0 -> negative
9> end.
** exception error: no true branch found when evaluating an if expression
```

Ruby や Io と違って，Erlang の if は関数なので，いずれかの文が必ず真にならなければならない．各ケースは値を返す必要がある．どうしても else 句に相当する部分が欲しければ，次のように最後のガードを true にすればよい．

```
9> if
9>   X > 0 -> positive;
9>   X < 0 -> negative;
9>   true  -> zero
9> end.
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

制御構文はこれだけだ．これから高階関数とパターンマッチングを使えばもっといろいろなことができるので，制御構文はこのくらいにして関数型プログラミングの世界に飛び込むことにしよう．具体的には，高階関数を作成し，それを使ってリストを処理する．また，より難しい問題を関数を使って解く方法を徐々に学んでいく．

無名関数

高階関数とは，戻り値として関数を返すか，引数として関数を受け取る関数のことだった．Ruby では，高階関数の代わりにコードブロックを使う．とりわけ，コードブロックを渡してリストを繰り返し処理するというやり方が重要だった．Erlang では，任意の関数を変数に代入し，その変数を他のデータ型と同じように受け渡しすることができる．

これらの概念のなかには既に説明済みのものもあるが，Erlang での基礎固めをある程度行ってから，高レベルの抽象化を構築していくことにしよう．まずは無名関数からだ．関数を変数に代入する方法を以下に示す．

```
16> Negate = fun(I) -> -I end.
#Fun<erl_eval.6.13229925>
17> Negate(1).
-1
18> Negate(-1).
1
```

16 行目に新しいキーワード `fun` が登場する．このキーワードが無名関数を定義する．この例では，無名関数が単一の引数 `I` を受け取り，その負の値`-I` を返す．この無名関数を `Negate` という変数に代入している．確認しておくが，`Negate` は無名関数によって返された値ではなく，無名関数そのものである点に注意してほしい．

ここで重要な考え方が 2 つ登場している．1 つ目は，関数を変数に代入している点だ．この概念によって，「振る舞い」を他のデータとまったく同じように受け渡すことができる．2 つ目は，変数に代入された関数を，引数リストを指定するだけで簡単に呼び出せる点だ．動的型付けにも注意してほしい．関数の戻り値の型について心配する必要がないため，Scala などに見られるうるさい構文を使わずに済む．欠点は，こうした無名関数で実行時エラーが発生する可能性があることだ．この制限を埋め合わせる方法を後でいくつか紹介する．

では学んだばかりのこの機能を使ってみよう．無名関数を使って，Ruby で登場した `each`，`map`，`inject` の概念を実装してみる．

リストと高階関数

既に見たように，リストとタプルは関数型プログラミングのキモだ．最初の関数型言語がリストから始まり，すべてがリストを基盤として構築されているのも偶然ではない．このセクションでは高階関数をリストに適用してみよう．

リストに関数を適用する

ここまでの説明でイメージはかなりはっきりしてきたはずだ．ここでは関数を使ってリストを管理してみる．`foreach` のようにリストを繰り返し処理するものもあれば，

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:03:33 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

filter や map のように，リストをフィルタリングして，または他の関数にマッピングして返すものもある．さらには foldl や foldr のように，中間結果を畳み込みながらリストを処理していくものもある（これは Ruby の inject や Scala の FoldLeft に似ている）．コンソールを再起動し，リストをいくつか定義して，さっそく試してみよう．

最初に，基本的な繰り返しを行ってみる．lists:foreach メソッドは，1 つの関数と 1 つのリストを引数として受け取る．引数に指定する関数は無名関数でも構わない．

```
1> Numbers = [1, 2, 3, 4].
[1,2,3,4]
2> lists:foreach(fun(Number) -> io:format("~p~n", [Number]) end, Numbers).
1
2
3
4
ok
```

2 行目は少し分かりづらいので，詳しく説明しよう．まず lists:foreach という関数を呼び出す．第 1 引数は無名関数 fun(Number) -> io:format("~p~n", [Number]) end だ．この関数は 1 つの引数をとり，渡された引数の値を io:format 関数を用いて出力する[*4]．第 2 引数は Numbers，すなわち 1 行目で定義したリストだ．次のように関数を別に定義しておけば分かりやすくなる．

```
3> Print = fun(X) -> io:format("~p~n", [X]) end.
```

これで Print が io:format 関数にバインドされる．次のようにコードを簡素化できる．

```
8> lists:foreach(Print, Numbers).
1
2
3
4
ok
```

基本的な繰り返しは以上だ．次に，マッピングを行う関数に進もう．マッピング関数は Ruby の collect と同じように動作する．すなわち，リストの各値を関数に渡して，関数で処理された結果のリストを生成する．lists:foreach と同様，lists:map は 1 つの関数と 1 つのリストを引数として受け取る．では，map を使って上記の数値リストの各値を 1 だけ増分してみよう．

```
10> lists:map(fun(X) -> X + 1 end, Numbers).
[2,3,4,5]
```

簡単だ．今度は fun(X) -> X + 1 end が無名関数だ．これによって各値を 1 だけ増分し，lists:map がその結果値のリストを生成する．

マップの定義は非常に簡単だ．

```
map(F, [H|T]) -> [F(H) | map(F, T)];
```

[*4] ~p はプリティプリント用，~n は改行，[Number] は出力する引数のリストだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
map(F, [])    -> [].
```

あるリストに対して F をマップすると，ヘッドが F(head)，テールが map(F, tail) のリストが返される．後でリスト内包表記について説明するとき，マップをもっとコンパクトにしたバージョンを紹介する．

次はフィルタリングだ．リストはブール値を用いてフィルタリングできる．まず次のような無名関数を定義して，Small に代入する．

```
11> Small = fun(X) -> X < 3 end.
#Fun<erl_eval.6.13229925>
12> Small(4).
false
13> Small(1).
true
```

これで，この関数を引数に指定してリストをフィルタリングできる．lists:filter 関数は，Small を満たす，すなわち 3 より小さいすべての要素からなるリストを新しく生成する．

```
14> lists:filter(Small, Numbers).
[1,2]
```

見てのとおり，Erlang ではこのようなコードを楽に書ける．Small 関数をリストに適用した結果を，all や any で判定することもできる．lists:all はリスト内のすべての要素がフィルタ条件を満たしている場合にのみ true を返す．

```
15> lists:all(Small, [0, 1, 2]).
true
16> lists:all(Small, [0, 1, 2, 3]).
false
```

lists:any は，リスト内のいずれかの要素がフィルタ条件を満たしていれば true を返す．

```
17> lists:any(Small, [0, 1, 2, 3]).
true
18> lists:any(Small, [3, 4, 5]).
false
```

空リストだとどうなるだろうか．

```
19> lists:any(Small, []).
false
20> lists:all(Small, []).
true
```

予想どおり，all は true を返し（つまり，リスト内のすべての要素（といっても空だが）がフィルタ条件を満たしている），any は false を返す（空リスト内にはフィルタを満足する要素は存在しない）．空リストの場合は，どのようなフィルタを用いても結果は変わらない．

また，リストのヘッド部分にあってフィルタ条件を満たすすべての要素のリストを作成したり，リストの先頭部分にあってフィルタ条件を満たすすべての要素を破棄したり

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

することもできる．

```
22> lists:takewhile(Small, Numbers).
[1,2]
23> lists:dropwhile(Small, Numbers).
[3,4]
24> lists:takewhile(Small, [1, 2, 1, 4, 1]).
[1,2,1]
25> lists:dropwhile(Small, [1, 2, 1, 4, 1]).
[4,1]
```

これらのテスト関数は，メッセージのヘッダ部分を処理したり破棄したりするのに便利だ．リストを処理する関数を駆け足で見てきたが，最後に foldl と foldr を紹介しておく．

foldl

この概念については既に知っているのではないかと思う．マトリックスの主人公ネオ並みに優秀で，これらの関数を習得している方は，基本的な例を読んで先に進んでほしい．ただ foldl は，人によっては習得に時間がかかるので，少し違った方法で説明してみることにする．

これらの関数は，リストの各項に対して関数を適用した結果を畳み込むときに便利だ．引数の 1 つがアキュムレータとして機能し，残りの引数が各要素を表す．lists:foldl は，関数，アキュムレータの初期値，およびリストを引数として受け取る．

```
28> Numbers.
[1,2,3,4]
29> lists:foldl(fun(X, Sum) -> X + Sum end, 0, Numbers).
10
```

少し単純化するために，無名関数を変数に代入し，各変数の名前も分かりやすいものに変えてみよう．

```
32> Adder = fun(ListItem, SumSoFar) -> ListItem + SumSoFar end.
#Fun<erl_eval.12.113037538>
33> InitialSum = 0.
0
34> lists:foldl(Adder, InitialSum, Numbers).
10
```

随分良くなった．中間合計を保持し続けるわけだ．SumSoFar と，Numbers リストの各数値を 1 つずつ関数 Adder に渡す．そのたびに合計値は大きくなる．lists:foldl 関数は，中間合計を覚えていて，それを Adder に戻す．最後に，最終合計を返すというわけだ．

ここまで，既存のリストを処理する関数について見てきたが，リストを 1 度に 1 要素ずつ作り出す方法については説明しなかった．次に視点を変えて，リストを作り出す様子について見てみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

高度なリストの概念

これまで紹介したリストの概念はすべて，他の言語で見たことのある概念を拡張したものだ．しかしリストはもう少し高度な扱い方ができる．これまで，リストを作り出す方法についてはまだ説明していなかった．簡単なコードブロックによる極めて基本的な抽象化を行っただけだった．

リストの構築

可変状態なしでリストを作り出すのは，一見難しく思えるかもしれない．Ruby や Io では，リストにそのまま要素を追加していく．しかし別の方法もある．要素が追加された新しいリストを返す方法だ．リストには通常，頭から順に要素を追加する．ここでは `[H|T]` 構文をマッチの右側で使用する．次のプログラムは，リスト構築手法を用いてリストの各要素を 2 倍している．

▶ code/erlang/double.erl

```
-module(double).
-export([double_all/1]).

double_all([]) -> [];
double_all([First|Rest]) -> [First + First|double_all(Rest)].
```

このモジュールは 1 つの関数 `double_all` をエクスポートしている．この関数には 2 つの異なる節がある．最初の節は，空リストに対して `double_all` を適用すると空リストが返されることを示す．このルールによって再帰が終了する．

2 番目のルールは `[H|T]` 構文を，関数定義部分だけでなく，マッチの述語部分でも使っている．マッチの左側に `[First|Rest]` を指定する例については既に見てきた．これにより，リストを先頭要素と残りの部分に分割することができる．

これをマッチの右側に指定すると，リストの分割ではなくリストの構築が行われる．上の例の `[First + First|double_all(Rest)]` は，`First + First` を先頭要素とし，`double_all(Rest)` を残りの部分とするリストを構築するという意味だ．

この関数は通常どおりコンパイルして実行できる．

```
8> c(double).
{ok,double}
9> double:double_all([1, 2, 3]).
[2,4,6]
```

コンソール上で，|を使った別のリスト構築方法を見てみよう．

```
14> [1| [2, 3]].
[1,2,3]
15> [[2, 3] | 1].
[[2,3]|1]
16> [[] | [2, 3]].
[[],2,3]
17> [1 | []].
[1]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

驚くようなことは何もないはずだ．第2引数はリストでなければならない．左側には何を書いても，新しいリストの先頭要素として追加される．

では，Erlangの高度な概念であるリスト内包表記について見ていくことにする．リスト内包表記はこれまでに説明した概念を組み合わせたものだ．

リスト内包表記

関数型言語で最も重要な関数の1つといえばmapだ．mapを使えばリストを変形させることができる．ちょうど『マトリックス』のエージェントたちが姿を変えるように．この機能はとても重要なので，Erlangでは，コンパクトでなおかつ複数の変換を同時に行える，より強力な形式を用意している．

コンソールを再起動して，まずは従来の方法でマッピングを行ってみる．

```
1> Fibs = [1, 1, 2, 3, 5].
[1,1,2,3,5]
2> Double = fun(X) -> X * 2 end.
#Fun<erl_eval.6.13229925>
3> lists:map(Double, Fibs).
[2,2,4,6,10]
```

最初に，Fibsという数値のリストと，渡されたものを2倍にするDoubleという無名関数を定義している．その後，lists:mapを呼び出して，各要素に対してDoubleを呼び出し，得られた結果からリストを構築している．素晴らしい方法だが，この方法は頻繁に使われるため，Erlangではこれと同じことを行うもっとコンパクトな構文を用意している．この構文はリスト内包表記と呼ばれる．上の例をリスト内包表記で書くと，次のようになる．

```
4> [Double(X) || X <- Fibs].
[2,2,4,6,10]
```

文章にすると「Fibsというリストの各要素Xに対して，Xの Doubleを計算する」となる．お望みなら中間変数を使わずに直接書いてもよい．

```
5> [X * 2 || X <- [1, 1, 2, 3, 5]].
[2,2,4,6,10]
```

考え方は同じだ．「[1, 1, 2, 3, 5]というリストの各要素Xに対して，X * 2を計算する」となる．より高度なリスト内包表記を構築してみよう．まずはmapのコンパクトな定義からだ．

```
map(F, L) -> [ F(X) || X <- L].
```

文章にすると「関数Fに対してリストLをマッピングすることにより，Lの各メンバであるXについてF(X)のコレクションを生成する」となる．ではリスト内包表記を使って，商品，数量，価格の一覧を操作してみよう．

```
7> Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}].
[{pencil,4,0.25},{pen,1,1.2},{paper,2,0.2}]
```

1ドルにつき8セントの消費税を加算する必要があるとする．簡単なリスト内包表記を

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

追加して買い物かごの中の商品に消費税を付けることができる．

```
8> WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
8>   {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

これまでに学習した Erlang の概念はすべて当てはまる．ここでもパターンマッチングが行われているのだ．文章にすると「Cart というリストからタプル{Product, Quantity, Price}を 1 つずつ取り出して，Product，Price，Quantity および消費税（Price * Quantity * 0.08）からなるタプルのリストを返す」となる．このコードは筆者には本当に美しく見える．このシンタックスによって，文字どおり要求に応じて自由にリストの形を変えることができる．

もう 1 つ例を挙げよう．手持ちのカタログに手を加えて，得意客向けに 50% 値引きしたカタログを作りたいとする．カタログの内容は次のような感じになる．まず，上記の買い物かごの商品カタログから数量を除去したカタログを取り出す．

```
10> Cat = [{Product, Price} || {Product, _, Price} <- Cart].
[{pencil,0.25},{pen,1.2},{paper,0.2}]
```

文章にすると「Cart リストから Product と Price（2 番目の属性である数量は無視）のタプルを 1 つずつ取り出して，Product と Price のタプルを返す」となる．これに値引き計算を加える．

```
11> DiscountedCat = [{Product, Price / 2} || {Product, Price} <- Cat].
[{pencil,0.125},{pen,0.6},{paper,0.1}]
```

コンパクトで，読みやすく，強力なシンタックスだ．美しい抽象化が行われている．

実を言うと，これでもリスト内包表記のパワーのほんの一部をお見せしたにすぎない．完全な形式はもっと強力だ．

- リスト内包表記は [式 || 節 1，節 2，...，節 N] という形式を持つ．
- リスト内包表記には任意の数の節を含めることができる．
- 節には，ジェネレータまたはフィルタを指定できる．
- フィルタには，ブール式またはブール値を返す関数を指定できる．
- Match <-List という形式のジェネレータは，左辺のパターンを右辺のリストの各要素とマッチングする．

それほど難しくはない．ジェネレータは追加操作，フィルタは削除操作を行う．ここにも Prolog の影響が色濃く出ている．ジェネレータは取り得る値を決定し，フィルタは指定された条件に従ってリストから要素を取り除く．以下にいくつか例を示す．

```
[X || X  <- [1, 2, 3, 4], X < 4, X > 1].
[2,3]
```

文章にすると「[1, 2, 3, 4] の各要素 X のうち，4 より小さく 1 より大きいものを返す」となる．複数のジェネレータを指定することもできる．

```
23> [{X, Y} || X  <- [1, 2, 3, 4], X < 3, Y <- [5, 6]].
[{1,5},{1,6},{2,5},{2,6}]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
24>
```

この例では，[1，2，3，4] から取り出した値 X のうち 3 より小さいものと，[5，6] から取り出した値 Y を組み合わせてタプル{X，Y}を作っている．X が 2 つ，Y が 2 つ得られる．Erlang は，両者の直積を返す．

これで 2 日目は終わりだ．ここまでは，Erlang を使ってシーケンシャルプログラミングを行ってきた．まとめを行った後，演習問題をやってみよう．

2 日目で学んだこと

2 日目では，Erlang の式やライブラリについては深入りしなかったが，ここまでで関数型プログラムを書くために必要な情報は十分に身についている．最初にいくつかの必須の制御構文について説明し，その後駆け足で（無名関数などを）説明した．

次に高階関数について説明した．高階関数をリスト内で用いて，リストの繰り返し処理，リストのフィルタリング，リストの変更を行った．Scala の場合と同様に foldl を使った畳み込み計算についても学んだ．

最後に高度なリスト概念について学んだ．リストを先頭要素と残りの要素に分解するには，[H|T] をマッチの左側に指定する．リストを構築するには，[H|T] をマッチの右側に指定するか，単独で使う．次にリスト内包表記を学んだ．この表記は，ジェネレータとフィルタを使ってリストを簡単に変形できる，エレガントで強力な抽象化だ．

シンタックスは一長一短だ．Erlang の動的型付けのおかげで，高度な概念も簡潔なコードで書くことができる．その一方で，case や if の各節の後にセミコロンを付ける必要があるなど，ぎこちない部分もあった．

3 日目では，Erlang の核心ともいうべき部分，並行プログラミングに迫ってみよう．

セルフスタディ 2 日目

試してみよう：

- [{erlang, "a functional language"}, {ruby, "an OO language"}] のようなキーワード値タプルのリストを考える．リストとキーワードを受け取り，そのキーワードに対応する値を返す関数を書け．
- [{item quantity price}, ...] という形式の買い物リストを考える．このとき，[{item total_price}, ...] という形式の items のリストを構築するリスト内包表記を書け．ただし，total_price は，quantity×price とする．

ボーナス問題：

- 長さ 9 のリストまたはタプルとして表現された三目並べのボードを読み込むプログラムを書け．勝者が確定している場合はその勝者（x または o のどちらか）を，これ以上指し手がない場合は cat を，勝負がついていない場合は no_winner を返すようにせよ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

6.4　3 日目：赤い錠剤（The Red Pill）

たいていの人は聞いたことがあるだろう．マトリックスでは，青い錠剤を飲むと何も知らないまま幸せに生きていくことができる．赤い錠剤を飲むと，現実を知らされる．現実は，ときとして苦痛を伴う．

IT 業界では，言ってみれば業界全体が青い錠剤をごくりと飲み込んでいるような状態だ．まるで何も知らない純真無垢でナイーブな子供のように．並行性の実装は難しい．だから普通は諦める．すると，可変状態を追加することになるため，複数のプログラムを並行に実行したとき衝突が起こる．関数やメソッドには副作用があるため，正確さを証明することも，結果を予測することもできない．スレッドは，パフォーマンスを高めるために，共有なしのプロセスではなく，状態共有プロセスで使用する．このため，各コードセクションを保護するために余分なコーディングが必要となる．

結果として大混乱をきたす．並行性は痛みを伴う．それは元々実装が難しいからではなく，我々が間違ったプログラミングモデルを適用してきたからだ．

この章の初めに，Erlang では，簡単なことが難しくなると書いた．副作用と可変状態を許さないという条件の下では，アプローチの方法を根本から変える必要がある．多くの人にとってまったく相容れないとも思える Prolog ベースのシンタックスも我慢しなければならないだろう．しかし今，その我慢が報われるときがきた．赤い錠剤，つまり並行性と信頼性は，あなたにとってキャンディのように思えるだろう．どのくらいおいしいキャンディなのか，じっくり見ていこう．

並行性を実現する基本プリミティブ

並行性を実現する基本プリミティブは，メッセージの送信（!を使用），プロセスの生成（spawn を使用），メッセージの受信（receive）の 3 つだ．3 日目の今日は，これら 3 つのプリミティブを用いて，メッセージの送受信を行い，それらを基本的なクライアント／サーバを実現するイディオムとしてまとめてみる．

基本的な受信ループ

最初に，翻訳プロセスから始めよう．プロセスにスペイン語の文字列を送信すると，英語に翻訳された応答が返される．一般には，プロセスを生成し，その中でメッセージを受信して処理するループを書く．

以下に基本的な受信ループを示す．

▶ code/erlang/translate.erl

```
-module(translate).
-export([loop/0]).

loop() ->
    receive
        "casa" ->
            io:format("house~n"),
            loop();

        "blanca" ->
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
            io:format("white~n"),
            loop();

        _ ->
            io:format("I don't understand.~n"),
            loop()

end.
```

これまでに紹介した例に比べて長いので，分割して説明しよう．先頭の 2 行は，translate というモジュールを定義し，loop という関数をエクスポートしているだけだ．次のブロックが loop() 関数の本体になる．

```
loop() ->
    ...
end.
```

本体の 3 箇所で loop() を呼び出しており，いずれも呼び出し元には戻っていないが，これで問題ない．Erlang では末尾再帰が最適化されるため，loop() が各 receive 節の最後の操作である限り，オーバーヘッドはほとんど発生しない．つまりこの例では，再帰呼び出しを使って永久ループさせている[*5]．では次の receive の説明に移ろう．

```
receive ->
    ...
```

この関数は別のプロセスからメッセージを受信する．receive は，Erlang の他のパターンマッチング構文（case や関数定義）と同じように動作する．receive の後には，複数のパターンマッチング構文が続く．次に個々のマッチング処理について説明する．

```
"casa" ->
    io:format("house~n"),
    loop();
```

これはマッチング節だ．シンタックスは case 文と同じだ．受信メッセージが文字列 "casa"にマッチすると，その後のコードが実行される．行の区切り文字には「,」を，節を終端するには「;」を指定する．この例では house という語を出力してから，loop を呼んでいる（loop は最後の関数呼び出しなのでスタックは消費しない）．他のマッチング節もすべて同じだ．

これで， receive ループのあるモジュールが出来上がった．ではこのモジュールを実際に使ってみよう．

プロセスの生成

まず，このモジュールをコンパイルする．

```
1> c(translate).
{ok,translate}
```

[*5] ［訳注］Erlang で無限ループを書く際のイディオムのひとつ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

プロセスを生成するには spawn 関数を使う．spawn 関数は関数を引数にとる．引数に指定された関数は，新しい軽量プロセス内で開始される．spawn はプロセス ID（PID）を返す．translate モジュール内の関数を spawn に次のようにして渡す．

```
2> Pid = spawn(fun translate:loop/0).
<0.38.0>
```

見てのとおり，Erlang は<0.38.0>というプロセス ID を返している．コンソールではプロセス ID が山括弧に囲まれて表示される．ここではプロセス生成の基本的な形式だけをお見せするが，他にもいくつか形式があることを知っておく必要がある．プロセスを名前で登録しておけば，他のプロセスが，例えば共通のサービスを，プロセス ID ではなく名前で見つけることができる．動的に変更できるようにしたいコード（ホットスワップ可能なコード）には，さらに別の形式を使える．リモートプロセスを生成する場合は spawn(Node, function) という形式を使う．こうしたさまざまな形式についての説明は本書の範囲外なので割愛する．

ここまでで，コードブロックを含むモジュールをコーディングし，そのモジュールを軽量プロセスとして spawn した．最後の手順では，このプロセスにメッセージを渡す．それには Erlang の 3 つ目の並行性基本プリミティブを用いる．

メッセージの送信

Scala の章で見たとおり，Erlang に分散メッセージを渡すには!演算子を用いる．この演算子の形式は Pid ! message だ．Pid は任意のプロセス識別子，message には任意の値（プリミティブ，リスト，タプルなど）を指定できる．いくつかメッセージを送信してみよう．

```
3> Pid ! "casa".
"house"
"casa"
4> Pid ! "blanca".
"white"
"blanca"
5> Pid ! "loco".
"I don't understand."
"loco"
```

行ごとに 1 つのメッセージが送信される．receive 節の io:format がメッセージを出力し，コンソールが式の戻り値（つまり送信したメッセージ）を出力している．

名前付きのリソースに分散メッセージを送信する場合は，node@server ! message という形式を用いる．リモートサーバのセットアップについては本書の対象外なので説明は割愛するが，ご自分で少し調べてもらえれば，分散サーバを簡単に稼働できる．

上の例では，基本プリミティブとそれらを組み合わせて基本的な非同期サービスを実現する方法を説明した．どこにも戻り値がないことに気づいたかもしれない．次のセクションでは同期メッセージを送信する方法について説明する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:05:50 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

同期メッセージ

一部の並行システムは，電話での会話と同じように非同期に動作する．つまり送信者はメッセージを送信し，相手からの応答を待たずに次のメッセージを送信する．一方，同期的に動作する並行システムもある．例えば Web がそうだ．ページの表示を要求し，応答を待っていると，Web サーバがページを返してくる．戻り値を出力するだけの翻訳サービスを，本当に翻訳された文字列を返すサービスに変えてみよう．

メッセージングモデルを非同期モデルから同期モデルに変更するには，次の 3 つの操作が必要になる．

- メッセージサービスの各 receive 節で，翻訳を要求するプロセスの ID と翻訳対象語のペアからなるタプルにマッチさせる．この ID を追加することで応答を返すことができる．
- 各 receive 節で，結果を出力する代わりに送信者に応答を返す．
- !プリミティブを使う代わりに，簡単な関数を書いて要求を送信し応答を待つ．

これで背景知識は得られたので，実装コードを見てみよう．

同期受信

まず receive 節にパラメータを追加する．つまりタプルを使う必要がある．パターンマッチングを使えば簡単だ．各 receive 節は次のようになる．

```
receive
    {Pid, "casa"} ->
        Pid ! "house",
        loop();
        ...
```

任意の要素（これは常にプロセス ID になるはず）と，その後に続く casa という語とマッチングを行う．マッチしたら，house という語を受信側に返して，ループの先頭に戻る．

このパターンでマッチする点に注意してほしい．これは receive の一般的な形式だ．すなわち送信側プロセスの ID がタプルの最初の要素になる．それ以外の点で唯一大きく異なるのは，結果を出力するのではなく送信する点だ．メッセージの送信はもう少し複雑になる．

同期送信

受信側と対をなす送信側では，メッセージを送信して，即座に応答を待つ必要がある．Receiver にプロセス ID が入ったとして，同期メッセージの送信は次のようになる．

```
Receiver !  "message_to_translate",
    receive
        Message -> do_something_with(Message)
    end
```

メッセージは頻繁に送信するため，サーバに対する要求をカプセル化することでサービスを単純化する．この例では，単純化された簡単なリモートプロシージャコールは次

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

のようになる．

```
translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

以上をすべて組み合わせると，並行プログラムが出来上がる．と言っても，ほんの少し複雑になっただけだ．

▶ code/erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa"} ->
            From ! "house",
            loop();

        {From, "blanca"} ->
            From ! "white",
            loop();

        {From, _} ->
            From ! "I don't understand.",
            loop()

end.

translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

使い方は次のとおり．

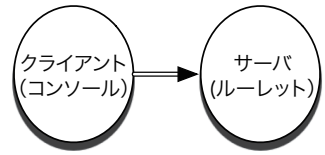
```
1> c(translate_service).
{ok,translate_service}
2> Translator = spawn(fun translate_service:loop/0).
<0.38.0>
3> translate_service:translate(Translator, "blanca").
"white"
4> translate_service:translate(Translator, "casa").
"house"
```

単純にコードをコンパイルし，ループを生成してから，作成したヘルパー関数を介して同期サービスを要求する．見てのとおり，Translator プロセスは指定した単語の翻訳結果を返すようになった．同期メッセージを受信するようになったわけだ．

以上で，基本的な receive ループの構造を理解できたと思う．各プロセスにはメールボックスが用意されている．receive 構文は，キューからメッセージを取り出して，特定の関数とマッチしたら，その関数を実行する．プロセスは，互いにメッセージパッシン

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 6.1 簡単なクライアント／サーバ設計

グによってやり取りする．Armstrong 博士が Erlang を真のオブジェクト指向言語と呼んだのは決して間違いではない．Erlang は，メッセージパッシングと振る舞いのカプセル化の仕組みを備えているのだ．可変状態と継承だけは実現されていないが，継承（およびその他の機能）は高階関数を用いてシミュレートできる．

ここまでは，エラー回復機能のない，基本的で，あまり面白みのない条件を想定してきた．Erlang には例外チェック機能が用意されてはいるが，ここでは別のエラー処理の方法を紹介してみたい．

プロセスのリンクによる信頼性の実現

このセクションでは，プロセスをリンクして信頼性を高める方法について見ていく．Erlang では，2 つのプロセスをリンクすることができる．プロセスは，自分が消滅するとき，自分とリンクしている相手プロセスに終了シグナルを送信する．そして，終了シグナルを受信したプロセスも終了する．

リンクされたプロセスの生成

リンクされたプロセスの動作原理を調べるために，簡単に終了する（死ぬ）プロセスを作成してみよう．ロシアンルーレットをコーディングしてみた．6 連発のピストルを使う．発射するには，1〜6 までの数字をピストルプロセスに送信する．間違った数字（弾の入ったシリンダーに相当する番号）を入力すると，プロセスは消滅する．以下にコードを示す．

▶ code/erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n"), exit({roulette,die,at,erlang:time()});
        _ -> io:format("click~n"), loop()
end.
```

実装は非常に簡単だ．メッセージループを作成する．3 にマッチすると，`io:format("bang~n"), exit({roulette,die,at,erlang:time()});` というコードが実行され，プロセスは消滅する．それ以外の番号の場合は，メッセージを出力し，

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ループの先頭に戻る．

これは簡単なクライアント／サーバ・プログラムにほかならない．クライアントはコンソール，サーバは roulette プロセスだ（図 6.1「簡単なクライアント／サーバ設計」(前ページ)を参照）．

このプログラムを実行すると，次のようになる．

```
1> c(roulette).
{ok,roulette}
2> Gun = spawn(fun roulette:loop/0).
<0.38.0>
3> Gun ! 1.
"click"
1
4> Gun ! 3.
"bang"
3
5> Gun ! 4.
4
6> Gun ! 1.
1
```

問題は，3 を受信した後でピストルプロセスが死ぬので，それ以上メッセージを送信しても何も起こらない点だ．プロセスが生きているかどうか調べることもできる．

```
7> erlang:is_process_alive(Gun).
false
```

確かにプロセスは死んでいる．亡きがらの回収を待つばかりだ．このプログラムをもう少し改良してみよう．プロセスが死んでいるかどうかを知らせるモニタを書いてみる．モニタというより検死官（coroner）といったほうがよいかもしれない．知りたいのは，死んでいるかどうかだけなのだから．

コードは次のようになる．

▶ code/erlang/coroner.erl

```
-module(coroner).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
    receive
        {monitor, Process} ->
            link(Process),
            io:format("Monitoring process.~n"),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p.", [From, Reason]),
                      io:format("Start another one.~n"),
            loop()
        end.
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

いつものように receive ループを作成する．最初に，exit をトラップするプロセスとして，このプログラムを登録する必要がある．そうしないと EXIT メッセージが受信されない．

次に受信内容を処理する．receive には 2 種類のタプルを指定する．monitor アトムで始まるタプルと，'EXIT' で始まるタプルだ．それぞれについて詳しく見ていこう．

```
{monitor, Process} ->
    link(Process),
    io:format("Monitoring process.~n"),
    loop();
```

このコードは，検死官プロセスを，Process という PID を持つ任意のプロセスにリンクする．spawn_link を使えば，リンクを設定した状態でプロセスを生成できる．監視対象のプロセスは，死ぬときに exit メッセージをこの検死官に送信する．では次に，エラーをトラップする部分に進もう．

```
{'EXIT', From, Reason} ->
    io:format("The shooter died. Start another one.~n"),
    loop()
end.
```

このコードは，exit メッセージにマッチする．'EXIT' の後に From（終了するプロセスから送られてきた PID），Reason（終了する理由）が続く 3 要素のタプルを使う．マッチしたら，終了するプロセスの PID と終了する理由を出力する．実行時の流れは次のようになる．

```
1> c(roulette).
{ok,roulette}
2> c(coroner).
{ok,coroner}
3> Revolver=spawn(fun roulette:loop/0).
<0.43.0>
4> Coroner=spawn(fun coroner:loop/0).
<0.45.0>
5> Coroner ! {monitor, Revolver}.
Monitoring process.
{monitor,<0.43.0>}
6> Revolver ! 1.
click
1
7> Revolver ! 3.
bang.
3
The shooter <0.43.0> died with reason
{roulette,die,at,{8,48,1}}. Start another one.
```

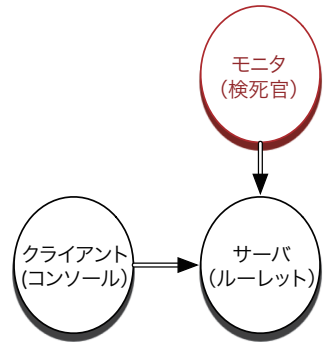
今度は，単なるクライアント／サーバよりは高度になった．監視プロセス（図 6.2「モニタを追加する」(次ページ)）を追加したので，プロセスが死ぬと分かる．

検死官から医者へ

さらに改良してみよう．ピストルを登録すると，ゲームプレーヤーはピストルの PID を知る必要がなくなる．これにより，ピストルの生成を検死官に任せることができる．し

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 6.2　モニタを追加する

たがって，プロセスが死んだら検死官がプロセスを再起動できる．そうすれば，信頼性が向上し，報告されるエラーの数も減る．こうなると，検死官はただの検死官ではなく，医者になる．死者をよみがえらせる医者である．以下にそのコードを示す．

▶ code/erlang/doctor.erl

```
-module(doctor).
-export([loop/0]).

loop() ->
    process_flag(trap_exit, true),
    receive
        new ->
            io:format("Creating and monitoring process.~n"),
            register(revolver, spawn_link(fun roulette:loop/0)),
            loop();

        {'EXIT', From, Reason} ->
            io:format("The shooter ~p died with reason ~p.", [From, Reason]),
                      io:format(" Restarting. ~n"),
            self() ! new,
            loop()
        end.
```

receive ブロックは，今度は 2 つのメッセージにマッチする．new と EXIT の 2 つのタプルだ．どちらも検死官プロセスの場合と若干異なる．次の 1 行が，new ブロックのキモだ．

```
register(revolver, spawn_link(fun roulette:loop/0)),
```

内側から順に説明していこう．まず，spawn_link を用いてプロセスを生成している．この形式の spawn は，roulette プロセスが死ぬといつでも doctor に通知されるように，2 つのプロセスをリンクする．その後，PID を登録し，revolver アトムに関連付ける．これでユーザは，revolver ! message を用いて，このプロセスにメッセージを送

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

信できる．今度はPIDは必要ない．EXITをマッチングするブロックも賢くなっている．以下にコードを示す．

```
self() ! new,
```

自分自身にメッセージを送信し，新しいピストルを生成および登録している．ゲームの開始も簡単になった．

```
2> c(doctor).
{ok,doctor}
3> Doc = spawn(fun doctor:loop/0).
<0.43.0>
4> revolver ! 1.
** exception error: bad argument
     in operator  !/2
        called as revolver ! 1
```

予想どおり，プロセスをまだ生成していないためエラーが発生している．プロセスを生成および登録してみよう．

```
5> Doc ! new.
Creating and monitoring process.
new
6> revolver ! 1.
click
1
7> revolver ! 3.
bang.
3
The shooter <0.47.0> died with reason {roulette,die,at,{8,53,40}}.
    Restarting.
Creating and monitoring process.
8> revolver ! 4.
click
4
```

Doctor経由でrevolverを作成するという手順が追加された（医者がピストルを用意するのはいささか矛盾してはいるが）．ピストルとやり取りするには，GunのPIDではなく，revolverアトムを介してメッセージを送信する．実は，8行目の後に，新しいピストルを作成および登録している．全体の構成は，図6.2「モニタを追加する」(前ページ)に示したものと変わらない．ただ，doctorがcoronerよりも積極的な役割を果たすようになっただけだ．

さて，ざっとではあるが，Erlangを使えば堅牢な並行システムを極めて簡単に作れることを分かってもらえたのではないかと思う．紹介した例ではエラー処理のためのコードがまったくなかった．何かがクラッシュしたら，単純に再起動しただけだ．お互いを監視するモニタも比較的簡単に構築できる．実際，基本ライブラリには，障害時には自動的に再起動されるモニタサービスや，キープアライブ機能を構築するためのツールが豊富に用意されている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

3 日目で学んだこと

3 日目は，「Erlang で何ができるのか」についてかなり感じがつかめてきた．並行性プリミティブ send，receive，spawn から始めて，生来の非同期バージョンの翻訳プログラムを作成し，基本的なメッセージパッシングの動作原理を説明した．次に，送信と受信をカプセル化してリモートプロシージャコールをシミュレートする簡単なヘルパー関数を作成した．

次に，プロセスをリンクして，一方のプロセスが死んだとき，他方に通知する方法を示した．あるプロセスを別のプロセスで監視して，信頼性を高める方法についても学んだ．3 日目で紹介したシステムは耐障害性を備えたものではないが，そこで用いた考え方は，耐障害性を備えたシステムの構築に利用できる．Erlang の分散通信はプロセス間通信とまったく同じように動作する．異なるコンピュータ上で動作する 2 つのプロセスをリンクし，スタンバイプロセスからマスタープロセスを監視して，問題が発生したらスタンバイ側で処理を引き継ぐようにすることもできる．

では，学んだことを次の演習で実際に試してみよう．

セルフスタディ 3 日目

以下の演習は比較的簡単だが，ボーナス問題を追加しておいたので，少しだけ頑張って解いてみてほしい．

Open Telecom Platform（OTP）は，分散並行サービスの構築に必要なほとんどの機能を提供する強力なパッケージだ．

探してみよう：

- プロセスが死んだときに，そのプロセスを再起動する OTP サービス
- 簡単な OTP サーバを構築するためのドキュメント

試してみよう：

- `translate_service` を監視し，死んだら再起動するようにせよ．
- `Doctor` プロセスが死んだとき，自分自身で再起動するようにしてみよ．
- `Doctor` モニタのモニタを作成し，どちらかのモニタが死んだら再起動するようにせよ．

次のボーナス問題に答えるには少し調査が必要になる．

- メッセージのログをファイルに記録する基本的な OTP サーバを作成せよ．
- ネットワーク越しに動作するように `translate_service` を改良してみよ．

6.5　Erlang のまとめ

この章の冒頭で，Erlang は難しいことを簡単にし，簡単なことを難しくすると書いた．Prolog 形式のシンタックスは C 系のさまざまな言語に慣れている人にはなじみがないだろうし，関数型プログラミング自体にも難しい面がある．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

しかし Erlang は，新しいハードウェア・デザインの登場で並行プログラミングの重要性が増しているため，極めて重要になるであろうコア機能を備えている．一部の機能の裏には哲学がある．軽量プロセスは Java のスレッド／プロセス・モデルと正反対の考え方だ．「クラッシュさせろ」の哲学により，コードは大幅に簡素化されるが，仮想マシンレベルで，他のシステムには存在しない基本的なサポートが必要になる．主要な長所と短所をまとめておく．

長所

Erlang の売りは，なんといっても並行性と耐障害性だ．プロセッサの設計者が分散コアに注目している今，最先端のプログラミング技術も進化しなければならない．Erlang の強みは，現世代のプログラマが直面するであろう最も重要な領域で発揮される．

ダイナミックさと信頼性

Erlang は，何よりもまず信頼性を優先して設計されている．コアライブラリの信頼性はテスト済みであり，Erlang のアプリケーションの信頼性と可用性は最高レベルである．特筆すべきは，言語の設計者が，Erlang の高い生産性を実現している動的型付けを犠牲にすることなく，この高い信頼性を達成している点である．Erlang は，コンパイラによる見せかけのセーフティネットではなく，並行プロセスを確実かつシンプルにリンクする機能に依存している．筆者は，オペレーティングシステムが提供するその場しのぎの危うい機能に依存することなく，信頼性の高いモニタを構築できることに驚いた．

Erlang に見られるいくつかの妥協点は，説得力がありユニークだと思う．Java と JVM は，Erlang と同等のパフォーマンスまたは信頼性を実現できるような適切なプリミティブを提供していない．Erlang の場合，BEAM 上に構築されているライブラリもその哲学を反映しているので，信頼性の高い分散システムを比較的容易に構築できる．

軽量の共有なしプロセス

Erlang のもう 1 つの特長は，基盤となるプロセスモデルだ．Erlang のプロセスは軽量なので，プログラマはこれをよく使う．Erlang は不変性を強制する哲学に基づいて設計されているため，プログラマが構築するシステムも，衝突によるエラーが本質的に発生しにくいシステムとなる．メッセージパッシング・パラダイムと各種プリミティブによって，オブジェクト指向アプリケーションではまず実現できないレベルの独立性を持つアプリケーションを簡単にコーディングできる．

OTP（エンタープライズ向けライブラリ）

Erlang は，高い可用性と信頼性が要求される電話会社で開発されたため，それに応じたスタイルの開発をサポートするために 20 年にわたって開発されたライブラリの蓄積がある．主要なライブラリとして Open Telecom Platform（OTP）がある．監視されたキープアライブプロセスの構築，データベースへのリンク，分散アプリケーションの構築を簡単に行えるライブラリも用意されている．OTP には完全な機能を備えた Web サーバや，通信アプリケーションに接続するための多数のツールが用意されている．

これらのライブラリの素晴らしいところは，耐障害性，スケーラビリティ，トランザク

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ション整合性，ホットスワップなどの機能が組み込まれている点だ．したがって，これらの機能を実装する必要はない．ユーザはこれらの機能を利用した独自のサーバプロセスを構築できる．

クラッシュさせろ

並列プロセスを副作用なしで実現することで初めて，「クラッシュさせろ」という方針が機能する．プロセスは簡単に再起動できるので，個々のプロセスがクラッシュしても，いちいちその理由を特定する必要がないからだ．関数型プログラミングモデルによって，Erlang の分散戦略の利点はさらに強化される．

本書で紹介した他のすべての言語と同様，Erlang にも問題点はある．違うのは問題の性質だけだ．以下に示すのは，エージェント・スミスがあまり得意としない領域だ．

短所

Erlang を採用するときの基本的な問題は，ニッチ言語に深く根ざした問題だ．まず，大半のプログラマにとって，シンタックスがなじみのないものになっている．また関数型プログラミングパラダイムは非常に異質なので，広範な普及を望むには問題となる．最後に，優れた実装が JVM 上にはなく BEAM 上にある．それぞれについて少し詳しく見てみよう．

シンタックス

シンタックスの評価は，映画の評価と同じで，多分に主観的だ（好みの問題だ）．しかし，Erlang のシンタックスには，偏見のない人でも気になるような問題がいくつかある．以下に 2 つ例を挙げる．

面白いことに，Erlang の長所は，Erlang の基盤となった Prolog の性質からきているのだが，短所にも同じことが言える．大半のプログラマにとって，Prolog は分かりにくく，シンタックスは不格好で異質である．ちょっとしたシンタックスシュガーがあれば，だいぶとっつきやすくなるだろう．

この章では，`if` と `case` の 2 つの構文の問題点を指摘した．文の間に区切り記号を挿入するという構文規則は筋が通ってはいるが，`case`，`if`，`receive` といったブロックの中の文の順番を変えると区切り記号も変えないといけないので，実用的ではない．このような制限は必要ない．ほかにも，場合によっては数値の配列が文字列とみなされるなど，奇妙な点がある．これらを修正すれば，Erlang は格段に良くなるだろう．

Java との統合性

Prolog の影響もそうだが，JVM 上で動作しないというのも一長一短だ．最近，Erjang という JVM ベースの仮想マシンが使われるようになってきたが，最高の JVM に匹敵するレベルにはまだ達していない．JVM にはいらぬおまけがついてくる．例えば，Erlang のニーズには合わないプロセス／スレッド・モデルなどだ．とはいうものの，JVM 上で動作すると，豊富な Java ライブラリが利用でき，数十万におよぶサーバがデプロイ対象になりうるといった利点もある．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:06:14 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

終わりに

プログラミング言語の成功はちょっとしたことですぐに変わる．Erlang は，マーケティング上の重大な問題に直面しており，また，Lisp スタイルのプログラミングパラダイムと Prolog スタイルのシンタックスに Java のプログラマを引きつけるのは容易なことではない．Erlang は，しかるべきときにしかるべき方法でしかるべき問題を解いてくれるので，普及にはずみがついている．このネオとエージェント・スミスの対決では，五分五分でエージェント・スミスにも勝つチャンスがあると思う．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:11:02 under the license of University of Tokyo

第 7 章

Clojure

やるか，やらないかだ．やってみるではない．

ヨーダ

Clojure は，JVM 上で動作する Lisp である．Lisp は，理解しにくいが強力な特性を備えており，最も古く，同時に最も新しいプログラミング言語の 1 つである．多くの方言が登場し，Lisp を主流言語にしようと試みたが失敗に終わった．シンタックスとプログラミングモデルは，これまでに登場した方言を見る限り，普通の開発者が吸収するには難しすぎるようだ．しかし，Lisp には，再挑戦してみる価値のある特別な何かがあるらしく，新しい方言が今でも登場している．いくつかの一流大学のコンピュータプログラミング関連学科では，吸収力のある柔軟な若い頭脳に，まず Lisp から学ばせている．

Clojure は多くの点で，賢明なカンフーの達人であり，丘の上に立つ予言者であり，はたまた謎めいたジェダイの指導者のようでもある．ヨーダを思い浮かべてほしい．映画『スター・ウォーズ エピソード V：帝国の逆襲』[*1]では，ヨーダは目立たないキュートなキャラクタとして登場していた．彼の話し方は，たいていの場合，語順が逆で，理解しづらい．ちょうど，Lisp の前置記法のように `(understand me later you will)`．彼はあまりに小さすぎて重要人物であるとは思えない．ちょうど，括弧と記号しか用いない Lisp のシンタックス・ルールのように．しかし，ヨーダにはその見た目からは想像できない何かがあることがすぐに明らかになる．Lisp 同様，彼は年老いてはいるが，その知恵は歳を重ねるにつれ研ぎ澄まされ，砲火の中で試されてきた．Lisp のマクロと高階構文のように，他の者は習得できない内なる力を彼は備えている．プログラミング言語において，Lisp を源とするアイデアは数多い．詳しい解説に入る前に，Lisp についてざっと説明してから，Clojure の利点に話を移していこう．

7.1 Clojure とは

いろいろと書いたが，Clojure は結局，もう 1 つの Lisp 方言である．Lisp と同じ言語的な制限があるが，同時に，Lisp と同じ多数の長所も備えている．Clojure を理解するには，まず Lisp を理解する必要がある．

[*1] 『スター・ウォーズ エピソード V：帝国の逆襲』．ジョージ・ルーカス監督作品．1980 年．カリフォルニア州ビバリーヒルズ．20 世紀フォックス，2004 年．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Lisp について

Lisp は，今も使われている言語のなかでは，Fortran に次いで古い商用言語である．Lisp は関数型言語だが，純粋な関数型言語ではない．Lisp は LISt Processing の略だが，この名前の理由はすぐに分かる．Lisp には，いくつかの面白い性質がある．

- Lisp はリストの言語である．関数呼び出しでは，リストの最初の要素を関数として，残りの要素を引数として用いる．
- Lisp は自分自身のデータ構造体を用いてコードを表現する．Lisp の支持者はこのことをコードとしてのデータと呼ぶ．

この 2 つのアイデアを組み合わせると，メタプログラミングに理想的な言語が出来上がる．クラスベースのオブジェクト指向がよければ，クラスや名前付きメソッドを書くことができる．それらのオブジェクトをツリー状に関係づければ，基本的なオブジェクトモデルが出来上がる．プロトタイプベースがよければ，スロットを使ってデータと振る舞いに変化を持たせればよい．純粋な関数型の実装もできる．こうした柔軟性のおかげで，Lisp はあらゆるプログラミングパラダイムに合わせて変形させることができる．

Paul Graham 氏は，『ハッカーと画家』[Gra04] という書籍の中で，少人数の開発チームが Lisp とその強力なプログラミングモデルを使って，大企業を打ち負かしたという話を語っている[*2]．この開発チームは，Lisp のほうがプログラミング言語としてはるかに有利であることを確信していた．実際，彼らは，Lisp やその他の高レベル言語のスキルを必要とする仕事の求人を行っている新興企業をより警戒していた．

Lisp の主要な方言として，Common Lisp と Scheme がある．Scheme と Clojure は lisp-1 系の同じファミリーに属する方言であり，Common Lisp は lisp-2 系の方言だ．これらの方言ファミリーの主な違いは，名前空間の動作方法と関係がある．Common Lisp は関数と変数にそれぞれ異なる名前空間を使うが，Scheme はどちらにも同じ名前空間を使う．以上の Lisp 側の話を踏まえて，Java 側の話に移ろう．

JVM 上で動作

Lisp のすべての方言は，ユーザの要望に応えたものだ．Clojure の場合，最も重要な性質の 1 つが JVM だ．Scala の例からも分かるとおり，商業的に成功しているプラットフォームが使えるということは大変大きな差となって現れる．Clojure を使うのに，デプロイ担当者に Clojure サーバを買ってもらう必要はない．Clojure は比較的新しい言語だが，数万におよぶ Java ライブラリを利用して，必要なことはほとんど何でもできる．

この章全体を通して，呼び出し方法，使用するライブラリ，作成する成果物などに，JVM の影響が見てとれる．しかし，JVM から解放された自由な部分もある．Clojure は関数型なので，高度な概念をコードに適用できる．Clojure は動的型付けを採用しているので，コードがコンパクトで読みやすくなり，コーディングも楽しくなる．そして何より，Clojure は Lisp の豊富な表現力を備えている．

[*2] ［訳注］「普通のやつらの上を行け」http://practical-scheme.net/trans/beating-the-averages-j.html

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Clojure と Java はお互いを大いに必要としている．Lisp には Java 仮想マシンによってもたらされる市場が必要だし，Java コミュニティには，本格的なアップデートと楽しさの注入が必要だ．

並行処理に関する改良

Clojure の性質として最後に挙げるのはライブラリ・セットだ．Clojure は関数型言語であり，副作用なしの関数を重視している．しかし，どうしても可変状態を使いたいというユーザ向けに，多くの概念をサポートしている．トランザクションメモリは，トランザクション・データベースのように動作し，メモリに対する安全な並行アクセスを実現する．エージェントを使うと，変更可能なリソースに対するアクセスをカプセル化できる．これらの概念の一部については 3 日目で取り上げる．

前置きはこのくらいにして，さっそく Clojure の説明に入ろう．

7.2　1 日目：ルークの修行

映画『スター・ウォーズ』では，ジェダイ流の高度な訓練を受けるために，ルークがヨーダに弟子入りする．ルークは彼のもとで修行を始める．ちょうどルークのように，読者の皆さんは，関数型言語の修行を既に開始している．Ruby でクロージャを使い，Scala と Erlang で高階関数を卒業した．この章では，Clojure で，これらの概念のいくつかを使う方法を学ぶ．

まず，Clojure のホームページ `http://clojure.org/getting_started` にアクセスし，指示に従って，お使いのプラットフォームの好みの開発環境に Clojure をインストールしてほしい．筆者は現在 Clojure 1.2 のプレリリース版を使っているが，皆さんが本書を手にする頃には 1.2 が正式にリリースされているはずだ[*3]．最初に Java プラットフォームをインストールする必要があるが，最近では Java は大半のオペレーティングシステムにインストール済みだ．本書では，Clojure のプロジェクトと Java のコンフィギュレーションの管理に leiningen[*4]を使う．このツールを使うと，プロジェクトをビルドする際，クラスパスなどの Java 側の詳細な情報を気にせずに済む．インストールが完了したら，新しいプロジェクトを作成する．

```
batate$ lein new seven-languages
Created new project in: seven-languages
batate$ cd seven-languages/
seven-languages batate$
```

これで，repl と呼ばれる Clojure コンソールを起動できる．

```
seven-languages batate$ lein repl
Copying 2 files to /Users/batate/lein/seven-languages/lib
user=>
```

[*3] ［訳注］Clojure 1.2 は 2010 年 8 月 19 日にリリースされ，翻訳時点では Clojure 1.2.1 が最新の安定版である．

[*4] `http://github.com/technomancy/leiningen`

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

これでよし．leiningen は，背後で必須ファイルをインストールし，数個の Clojure Java アーカイブ（jar）とオプションを指定して Java を起動する．repl の起動方法はお使いのディストリビューションによって異なるが，この章では以降，単に「repl を起動する」と書くだけに留め，具体的な起動方法については触れない．

以上の作業が完了したら，基本的な機能を備えたコンソールが起動される．この章でコードを評価する場合は，この基本コンソールを使うか，Clojure がサポートされている任意の IDE またはエディタを使う．

少しコードを入力してみよう．

```
user=> (println "Give me some Clojure!")
Give me some Clojure!
nil
```

コンソールは動いているようだ．Clojure では，関数呼び出し全体を括弧で囲む．最初の要素は関数名，残りは引数だ．入れ子にすることもできる．簡単な数式で試してみよう．

基本的な関数の呼び出し

```
user=> (- 1)
-1
user=> (+ 1 1)
2
user=> (* 10 10)
100
```

簡単な算術演算だ．割り算はもう少し面白い．

```
user=> (/ 1 3)
1/3
user=> (/ 2 4)
1/2
user=> (/ 2.0 4)
0.5
user=> (class (/ 1 3))
clojure.lang.Ratio
```

Clojure には，*ratio* と呼ばれる基本データ型がある．この機能は，精度を失わないように計算を遅延させることができる優れものだ．もちろん望むなら浮動小数点数でも簡単に処理できる．余りも簡単に計算できる．

```
user=> (mod 5 4)
1
```

これは剰余演算子の短縮形だ．この記法は前置記法と呼ばれる．これまでに紹介した言語は中置記法を採用していた．つまり，`4 + 1 - 2` のように，演算子をオペランドの間に挿入する記法だ．たいていの人は，慣れているという理由で中置記法を好む．数学の式でも中置記法に見慣れている．それでも，少し練習すれば，前置記法の思考の流れに慣れるはずだ．数式をこのように書くのは少し不格好だが，問題なく動作する．括弧付きの前置記法には利点もある．次の式を見てほしい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

```
user=> (/ (/ 12 2) (/ 6 2))
2
```

あいまいさがまったくない．Clojure はこの文を括弧順に評価する．さらに，次の式を評価してみてほしい．

```
user=> (+ 2 2 2 2)
8
```

このように，お望みなら計算の要素を付け足すことも簡単にできる．この書き方は，引き算や割り算でも使える．

```
user=> (- 8 1 2)
5
user=> (/ 8 2 2)
2
```

上の例を中置記法で書くと，(8 - 1) - 2 および (8 / 2) / 2 となる．Clojure でも同じ演算子を一度に 2 回使って，(- (- 8 1) 2) および (/ (/ 8 2) 2) のように書くこともできる．次のように，簡単な演算子評価から至極便利な結果が得られることもある．

```
user=> (< 1 2 3)
true
user=> (< 1 3 2 4)
false
```

これはいい．リスト内の数値が順番に並んでいるかどうかを，1 つの演算子で確認できる．しかも，引数の数はいくつでもよい．

前置記法と複数引数リストの特殊な記法を除き，Clojure のシンタックスは極めて単純だ．型付けシステムについて少し調べてみよう．具体的には，強い型付けと強制型変換の 2 つだ．

```
user=> (+ 3.0 5)
8.0
user=> (+ 3 5.0)
8.0
```

Clojure は，我々に代わって，型を強制変換してくれる．一般には，Clojure は強い動的型付けをサポートしていることが分かる．もう少し焦点を絞って，Clojure の基本構築ブロックであるフォームについて見てみよう．

フォームは，構文の断片と考えることができる．Clojure はコードを解析するとき，プログラムをフォームと呼ばれる断片に分解する．これでコードをコンパイルするか，インタープリタで実行することができる．この章では，コードとデータを区別しない．というのは，Lisp では，両者は同じものだからだ．ブール値，文字，文字列，セット，マップ，ベクタはすべてフォームの例であり，本章のいたるところに出てくる．

文字列と文字

文字列は既に登場したが，もう少し詳しく見てみよう．文字列はダブルクオートで囲み，Ruby と同じように，C 形式のエスケープ文字が使える．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> (println "master yoda\nluke skywalker\ndarth vader")
master yoda
luke skywalker
darth vader
nil
```

驚くようなことは何もない．余談だが，ここでは，println に 1 つの引数を指定したが，引数の数は 0 個でも 2 個以上でも構わない．0 個なら空行が，2 個以上なら複数の値が連結されて出力される．

Clojure では，str 関数を用いて，他の型を文字列に変換できる．

```
user=> (str 1)
"1"
```

ターゲットが Java のオブジェクトであれば，str は Java の toString 関数を呼び出す．str は次のように複数の引数を取る．

```
user=> (str "yoda, " "luke,  " "darth")
"yoda, luke,  darth"
```

このため，Clojure の開発者は str を使って文字列を連結する．便利なことに，文字列以外の項を連結することもできる．

```
user=> (str "one: " 1 " two: " 2)
"one: 1 two: 2"
```

異なる型を連結することも可能だ．ダブルクオートの外で文字を表すには，次のように文字の前に\を付ける．

```
user=> \a
\a
```

個々の文字を str を使って連結することもできる．

```
user=> (str \f \o \r \c \e)
"force"
```

次のような比較をするとどうなるだろうか．

```
user=> (= "a" \a)
false
```

つまり，文字は長さ 1 の文字列ではないということだ．

```
user=> (= (str \a) "a")
true
```

ただし，文字は簡単に文字列に変換できる．文字列操作に関してはとりあえずこの辺にして，次にブール式の説明に移ろう．

ブール値と式

Clojure は，強く動的な型付けを採用している．動的型付けとは，型が実行時に評価されるという意味であることを思い出してほしい．既に，いくつかの型を実際に使ってい

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

るのを見てもらったが，もう少し焦点を絞って見てみよう．ブール値は式の結果だ．

```
user=> (= 1 1.0)
true
user=> (= 1 2)
false
user=> (< 1 2)
true
```

本書に登場する大半の言語と同じで，true はシンボルだが，同時に，他の何かでもある．Clojure の型は，基盤となる Java の型システムと統一されている．基盤となるクラスを取得するには，class 関数を用いる．ブール値のクラスは次のとおりだ．

```
user=> (class true)
java.lang.Boolean
user=> (class (= 1 1))
java.lang.Boolean
```

見てのとおり，JVM が顔をのぞかせている．この型戦略は，これから学習を進めるにしたがって，どんどん便利になる．評価結果としてのブール値は，多くの式で利用できる．以下に簡単な if を示す．

```
user=> (if true (println "True it is."))
True it is.
nil
user=> (if (> 1 2) (println "True it is."))
nil
```

Io と同様，if の第 2 引数としてコードを渡せる．都合の良いことに，Lisp ではデータをコードとして扱うことができる．次のようにコードを複数行に分割すれば見やすくなる．

```
user=> (if (< 1 2)
  (println "False it is not."))
False it is not.
nil
```

else を第 3 引数として追加できる．

```
user=> (if false (println "true") (println "false"))
false
nil
```

他にどのようなものがブール値として使えるか見てみよう．まず，Clojure では，何が nil として扱われるのだろうか．

```
user=> (first ())
nil
```

これは簡単だ．nil というシンボルが使われる．

```
user=> (if 0 (println "true"))
true
nil
user=> (if nil (println "true"))
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
nil
user=> (if "" (println "true"))
true
nil
```

0 と""は真だが，nil は真ではない．他のブール式は，必要になった時点で導入することにする．では，もう少し複雑なデータ構造体について見ていこう．

リスト，マップ，セット，ベクタ

すべての関数型言語と同様，リストやタプルなどの基本的なデータ構造体は強力だ．Clojure では，3 つの主要なデータ構造体として，リスト，マップ，ベクタがある．まずは，これまでにかなりの時間を割いて学んできたコレクション，すなわちリストから始めることにしよう．

リスト

リストは，要素の順序付きコレクションだ．リストを構成する要素は何でも構わないが，Clojure の慣用法では，コードにはリストを，データにはベクタを使う．ただしここでは混乱を避けるため，リストでデータを扱う方法を説明する．リストは関数として評価されるため，次のようには書けない．

```
user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer
    cannot be cast to clojure.lang.IFn (NO_SOURCE_FILE:0)
```

1，2，3 で構成されるリストを作成するには，次のどちらかを実行する必要がある．

```
user=> (list 1 2 3)
(1 2 3)
user=> '(1 2 3)
(1 2 3)
```

これで通常どおりリストを操作できる．2 番目の形式はクオートと呼ばれる．リスト用の 4 つの主要演算子として，first（ヘッド），rest（ヘッドを除いた残り），last（最終要素），cons（ヘッドとテールを与えて新しいリストを作る）がある．

```
user=> (first '(:r2d2 :c3po))
:r2d2
user=> (last '(:r2d2 :c3po))
:c3po
user=> (rest '(:r2d2 :c3po))
(:c3po)
user=> (cons :battle-droid '(:r2d2 :c3po))
(:battle-droid :r2d2 :c3po)
```

もちろん，これらを高階関数と組み合わせることもできるが，それについてはシーケンスが登場してから説明することにする．次に，リストの親類にあたるベクタの説明に移ろう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 12:11:02 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ベクタ

リスト同様，ベクタも，順序付きの要素コレクションだ．ベクタはランダムアクセス用に最適化されている．ベクタは，次のように角括弧で囲む．

```
user=> [:hutt :wookie :ewok]
[:hutt :wookie :ewok]
```

コードにはリストを，データにはベクタを使う．次のようにさまざまな要素を取得できる．

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (nth [:hutt :wookie :ewok] 2)
:ewok
user=> (nth [:hutt :wookie :ewok] 0)
:hutt
user=> (last [:hutt :wookie :ewok])
:ewok
user=> ([:hutt :wookie :ewok] 2)
:ewok
```

ベクタもやはり関数である点に注意してほしい．上の例の最後で，インデックスを引数として受け取っている．次のようにして，2つのベクタを組み合わせることができる．

```
user=> (concat [:darth-vader] [:darth-maul])
(:darth-vader :darth-maul)
```

replがベクタの代わりにリストを出力している点に気づいただろうか．コレクションを返す関数の多くは，シーケンスと呼ばれるClojureの抽象化機構を使う．シーケンスについては2日目で詳しく学ぶ．ここでは，Clojureがシーケンスを返し，それがreplではリストとして表示されているということだけを理解しておけばよい．

もちろん，ベクタのヘッドとテールを取得することもできる．

```
user=> (first [:hutt :wookie :ewok])
:hutt
user=> (rest [:hutt :wookie :ewok])
(:wookie :ewok)
```

これらの機能はどちらもパターンマッチングと組み合わせて使える．リストとベクタはどちらも順序付きだ．次に，順序なしのコレクションであるセットとマップの説明に移ろう．

セット

セットは，順序なしの要素コレクションだ．セットには固定された順序があるが，それは実装依存なので，その順序で並んでいるという前提で何か処理をしてはならない．セットは，次のように#{}で囲む．

```
user=> #{:x-wing :y-wing :tie-fighter}
#{:x-wing :y-wing :tie-fighter}
```

このセットを変数spacecraftに代入して操作してみる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> (def spacecraft #{:x-wing :y-wing :tie-fighter})
#'user/spacecraft
user=> spacecraft
#{:x-wing :y-wing :tie-fighter}
user=> (count spacecraft)
3
user=> (sort spacecraft)
(:tie-fighter :x-wing :y-wing)
```

順序のないセットとは別に，要素を受け取った順にかかわらずソートした順番で返すソート・セット（sorted set）というものもある[*5]．

```
user=> (sorted-set 2 3 1)
#{1 2 3}
```

次のように 2 つのセットをマージすることもできる．

```
user=> (clojure.set/union #{:skywalker} #{:vader})
#{:skywalker :vader}
```

差分を計算することもできる．

```
(clojure.set/difference #{1 2 3} #{2})
```

最後に，少し変わっているが便利な機能を紹介しておく．セットは関数でもあるのだ．#{:jar-jar, :chewbacca}は関数に操作される対象だが，同時に関数でもある．セットは，ある要素が自分のメンバかどうかを次のようにして判定する．

```
user=> (#{:jar-jar :chewbacca} :chewbacca)
:chewbacca
user=> (#{:jar-jar :chewbacca} :luke)
nil
```

セットを関数として使うと，その関数は，第 1 引数がセットのメンバであればその第 1 引数を返す．セットの基本的な説明は以上だ．次にマップの説明に移ろう．

マップ

ご存じのとおり，マップはキーと値のペアである．Clojure では，次のように，マップを中括弧で囲んで表現する．

```
user=> {:chewie :wookie :lea :human}
{:chewie :wookie, :lea :human}
```

これらはマップ，すなわちキーと値のペアだが，読みづらい．キーと値の数が奇数個になっていても見つけにくいため，次のようにエラーになる．

```
user=> {:jabba :hut :han}
java.lang.ArrayIndexOutOfBoundsException: 3
```

Clojure では，次のように，カンマを空白として使うことで，この問題を解決している．

[*5] ［訳注］リテラルのセットは clojure.lang.PersistentHashSet だが，sorted-set は clojure.lang.PersistentTreeSet を返す．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> {:darth-vader "obi wan", :luke "yoda"}
{:darth-vader "obi wan", :luke "yoda"}
```

Rubyのシンボル，あるいはPrologやErlangのアトムと同様，先頭に:が付いている単語がキーワードだ．Clojureには，このようにしてモノに名前を付けるための形式が2種類用意されている．すなわち，キーワードとシンボルだ．シンボルは他の何かを指しているが，キーワードは自分自身を指している．trueやmapはシンボルだ．キーワードは，map内のプロパティなど，ドメインエンティティに名前を付ける場合に使用する．これはErlangでアトムを使う場合と同じだ．

mentorsという名前のマップを定義してみよう．

```
user=> (def mentors {:darth-vader "obi wan", :luke "yoda"})
#'user/mentors
user=> mentors
{:darth-vader "obi wan", :luke "yoda"}
```

これで，キーを最初の値として渡すことで値を取得できる．

```
user=> (mentors :luke)
"yoda"
```

このようにマップも関数だが，次のようにキーワードもやはり関数だ．

```
user=> (:luke mentors)
"yoda"
```

関数:lukeは，マップ内で自分自身を探す．これは少し奇妙だが便利だ．Ruby同様，キーと値には任意のデータ型を使える．2つのマップをマージするにはmergeを用いる．

```
user=> (merge {:y-wing 2, :x-wing 4} {:tie-fighter 2})
{:tie-fighter 2, :y-wing 2, :x-wing 4}
```

両方のマップに同じキーが存在するときに値をどうやって合成するかを指定することもできる．

```
user=> (merge-with + {:y-wing 2, :x-wing 4} {:tie-fighter 2 :x-wing 3})
{:tie-fighter 2, :y-wing 2, :x-wing 7}
```

上の例では，キーx-wingに関連付けられた値4と3を+でマージしている．次のように，ある関連付け（マップ）に対して，新しいキーと値のペアで新しい関連付けを追加することもできる．

```
user=>(assoc {:one 1} :two 2)
{:two 2, :one 1}
```

ソート済みマップを作成することもできる．ソート済みマップは，任意の順序の項目を受け取り，それらをソートして出力する．次に例を示す．

```
user=> (sorted-map 1 :one, 3 :three, 2 :two)
{1 :one, 2 :two, 3 :three}
```

データに少しずつ構造を追加してきた．次に，振る舞いを追加する形式，つまり関数の説明に移ろう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

関数を定義する

関数は Lisp のすべての方言の中核をなす．関数を定義するには *defn* を用いる．

```
user=> (defn force-it [] (str "Use the force," "Luke."))
#'user/force-it
```

最も簡単な形式は (defn name [params] body) である．上の例では force-it という引数なしの関数を定義している．この関数は，2 つの文字列を単純に連結するだけだ．呼び出し形式は他の関数と変わらない．

```
user=> (force-it)
"Use the force,Luke."
```

必要なら，関数を説明する文字列を指定することもできる．

```
user=> (defn force-it
             "The first function a young Jedi needs"
             []
             (str "Use the force," "Luke"))
```

指定した説明文（ドキュメント）を呼び出すには，doc を使う．

```
user=> (doc force-it)
-------------------------
user/force-it
([])
  The first function a young Jedi needs
nil
```

引数を追加してみよう．

```
user=> (defn force-it
             "The first function a young Jedi needs"
             [jedi]
             (str "Use the force," jedi))
#'user/force-it
user=> (force-it "Luke")
"Use the force,Luke"
```

ところで，この doc 機能は，ドキュメント行が指定された他の任意の関数で利用できる．

```
user=> (doc str)
-------------------------
clojure.core/str
([] [x] [x & ys])
  With no args, returns the empty string. With one arg x, returns
  x.toString().  (str nil) returns the empty string. With more than
  one arg, returns the concatenation of the str values of the args.
nil
```

これで，基本的な関数が定義できるようになったので，パラメータリストの説明に移ろう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

バインディング

これまでに見た他の言語と同様，関数の仮引数に，関数に渡された実引数の値を代入することをバインディング（束縛）と言う．Clojureで気の利いた機能の1つに，引数の任意の部分にパラメータとしてアクセスできる機能がある．例えば，ベクタで表現された2点を表す次のような行があるとする．

```
user=> (def line [[0 0] [10 20]])
#'user/line
user=> line
[[0 0] [10 20]]
```

この行の最後の部分を取り出す関数は次のようにして書くことができる．

```
user=> (defn line-end [ln] (last  ln))
#'user/line-end
user=> (line-end line)
[10 20]
```

しかし，行全体は必要ない．パラメータを行の2番目の要素にバインドできたら便利だ．これはClojureを使えば次のように簡単にできる．

```
user=> (defn line-end [[_ second]] second)
#'user/line-end
user=> (line-end line)
[10 20]
```

この概念をデストラクチャリング（分配束縛）という．データ構造体を分解して，必要な部分だけを取り出しているわけだ．バインディングについて詳しく見てみよう．[[_ second]]の外側の括弧は，パラメータ・ベクタを定義している．内側の括弧は，リストまたはベクタの個々の要素にバインドすることを意味する．_とsecondは個々のパラメータだが，無視するパラメータには_を用いるのが慣用法である．文章にすると，「この関数のパラメータは，第1引数の最初の要素が_，第1引数の2番目の要素がsecondである」となる．

これらのバインディングはネスト化することもできる．例えば三目並べのボードがあるとき，その中央のマス目の値を返したいとする．このボードは次のように，3要素から成る3つの行として表現できる．

```
user=> (def board [[:x :o :x] [:o :x :o] [:o :x :o]])
#'user/board
```

このうち2行目の2番目の要素を選択したいわけだ．次のようにすればよい．

```
user=> (defn center [[_ [_ c _] _]] c)
#'user/center
```

美しい．基本的には，前と同じ概念をネスト化しただけである．分解して説明しよう．バインディングを行っているのは[[_ [_ c _] _]]だ．1つのパラメータを入力引数[_ [_ c _] _]にバインドする．このパラメータは，1番目の要素と3番目の要素（この例では三目並べの最上行と最下行にあたる）を無視するという意味だ．そして，中央の行だけに着目する．それが[_ c _]だ．これで，リストの中央の要素を取り出している．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> (center board)
:x
```

この関数は，2 つの方法で簡素化できる．まず，目的の引数の後にくるワイルドカード引数はすべて不要だ．

```
(defn center [[_ [_ c]]] c)
```

次に，デストラクチャリングは，引数リスト内だけでなく，let 文内でも起こすことができる．Lisp では let を使って変数を値にバインドする．let を使えば，center 関数のクライアントからデストラクチャリングを隠ぺいできる．

```
(defn center [board]
 (let [[_ [_ c]] board] c))
```

let は 2 つの引数をとる．第 1 引数は，バインドするシンボル（[[_ [_c]]]）とバインドされる値（board）から成るベクタだ．第 2 引数は，たいていその値を使った式だ（この例では単純に c を返している）．どちらの形式も結果は同じだ．どちらを選ぶかは，どこでデストラクチャリングを発生させたいかによる．以下に let を使った簡単な例をお見せするが，引数リスト内でも同じことができることを理解しておいてほしい．

マップをデストラクチャリングするには，次のようにする．

```
user=> (def person {:name "Jabba" :profession "Gangster"})
#'user/person
user=> (let [{name :name} person] (str "The person's name is " name))
"The person's name is Jabba"
```

マップとベクタを組み合わせることもできる．

```
user=> (def villains [{:name "Godzilla" :size "big"} {:name "Ebola" :size "small"}])
#'user/villains
user=> (let [[_ {name :name}] villains] (str "Name of the second villain: " name))
"Name of the second villain: Ebola"
```

ここでは，ベクタにバインドして，最初のマップをスキップし，2 番目のマップの名前を取得している．Lisp は，Prolog，そして Erlang にも影響を与えていることが分かる．デストラクチャリングは，単に，パターンマッチングの一形態にすぎない．

無名関数

Lisp では，関数は単なるデータである．コードは他の任意の種類のデータと同じなので，高階関数は完全に言語に組み込まれている．無名関数を使うと名前のない関数を作成できる．無名関数は，本書に登場するすべての言語の基本機能だ．Clojure では，無名関数を fn を用いて定義する．通常は名前を省略するため，(fn [parameters*] body) というような形式になる．例を挙げよう．

高階関数を使って，与えられた単語のリストに対して，単語の文字数のリストを構築してみよう．次のような人物名のリストがあるとする．

```
user=> (def people ["Lea", "Han Solo"])
#'user/people
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

1つの単語の長さを計算するには，次のように書く．

```
user=> (count "Lea")
3
```

名前の長さのリストを構築するには，次のように書く．

```
user=> (map count people)
(3 8)
```

この概念は前に出てきた．このコンテキストでは，map が高階関数だ．Clojure では，高階関数の概念は簡単に実現できる．というのは，関数はリストであり，他のリスト要素と何ら変わりないからだ．同じ構築ブロックを用いて，各人物名の長さの2倍を要素に持つリストを計算することもできる．

```
user=> (defn twice-count [w] (* 2 (count w)))
#'user/twice-count
user=> (twice-count "Lando")
10
user=> (map twice-count people)
(6 16)
```

この関数は非常にシンプルなので，次のように無名関数として書ける．

```
user=> (map (fn [w] (* 2 (count w))) people)
(6 16)
```

次のように，簡略化することもできる．

```
user=> (map #(* 2 (count %)) people)
(6 16)
```

簡略化された形式では，#によって無名関数を定義し，% がシーケンスの各項にバインドされる．#のことをリーダーマクロという．

無名関数は，今すぐには名前を必要としない関数を作成するための便利な道具である．無名関数は他の言語にも登場した．以下に，高階関数を用いてコレクションを処理する関数をいくつか紹介する．これらの関数では，共通のベクタ v を使う．

```
user=> (def v [3 1 2])
#'user/v
```

以下の例では，このリストをいくつかの無名関数で処理する．

apply

apply は，関数を引数リストに適用する．(apply f '(x y)) は，(f x y) と等価である．次に例を示す．

```
user=> (apply + v)
6
user=> (apply max v)
3
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

filter

filter 関数は，Ruby の find_all 関数と同じように動作する．対象をテストする関数を引数にとり，テストに合格した要素のシーケンスを返す．例えば，奇数の要素のみ，または 3 未満の要素のみを取得するには，次のように書く．

```
user=> (filter odd? v)
(3 1)
user=> (filter #(< % 3) v)
(1 2)
```

無名関数については，Clojure のシーケンスについて詳しく説明するときに詳しく見ていく．この辺りで一息入れて，Clojure の開発者である Rich Hickey 氏の話を聞くことにしよう．

Clojure の開発者 Rich Hickey 氏へのインタビュー

Rich Hickey 氏が本書の読者のためにいくつかの質問に答えてくれた．彼は Lisp の一方言である Clojure が他の方言よりも普及した理由に特に興味を持っているため，このインタビューはいつもより少し広範囲にわたっており，筆者自身楽しめた．皆さんにも楽しんでもらえれば幸いだ．

Bruce Tate　Clojure を開発した理由を聞かせてください．

Rich Hickey　私はプログラマとして，JVM や CLR などの業界標準のプラットフォーム上で動作する，ほぼ関数型で，拡張可能，動的，なおかつ堅牢な並行性を備えた言語が欲しかったのですが，見つからなかったのです．

Bruce Tate　Clojure のどの部分が最も気に入っていますか．

Rich Hickey　データ構造とライブラリの抽象化およびシンプルさを重視した点でしょうか．この 2 点は関連し合っていると思います．

Bruce Tate　一から作り直せるとしたら，どの部分を変更しますか？

Rich Hickey　数値に対する異なるアプローチを考えるでしょうね．JVM では，ボックス化された数値が確実に弱点になっていますから．この点については今取り組んでいるところです．

Bruce Tate　Clojure を用いたソリューションで，最も面白いと思ったものを教えてください．

Rich Hickey　FlightCaster[*6]（飛行機の便の遅れをリアルタイムで予測するサービス）は，Clojure のさまざまな機能を活用していると思います．具体的には，マクロによるシンタックスの抽象化を用いた機械学習と統計的推論ビットのための DSL，Hadoop や Cascading などのインフラストラクチャを利用するための Java との相互運用性などです．

[*6] http://www.infoq.com/articles/flightcaster-clojure-rails

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Bruce Tate　多くの Lisp の方言が失敗に終わっているなかで，Clojure が普及した理由は何だと思いますか？

Rich Hickey　それは重要な質問です．Lisp の主要な方言（Scheme と Common Lisp）が失敗に終わった（「その使命を果たすことができなかった」）と言えるのかどうかは私には分かりません．Scheme は，計算の要点をとらえた非常にコンパクトな言語を実現しようとしたものですし，Common Lisp は，研究向けに使われている Lisp のさまざまな方言を標準化しようとしたものです．どちらも企業の開発者が使う汎用プログラミング言語という実用的な道具としては確かに普及しませんでしたが，そもそもどちらもそうした汎用言語として設計されたわけではありませんから．
一方，Clojure は，企業の開発者向けに，汎用プログラミングのための実用的な道具として設計されており，既存の古い Lisp 系言語に実用性という新たな目標を追加した言語になっています．Clojure ではチームによる共同開発のほうが効率が良く，他の言語との連携も考慮され，Lisp 系の言語が抱えていた問題も解決しています．

Bruce Tate　Clojure では，なぜチーム開発のほうが効率が良いのですか．

Rich Hickey　一部の Lisp 系言語が個々の開発者の能力を最大化することを目的としており，それはそれで意義のあることですが，Clojure では，開発はチーム作業だと認識しています．例えば，ユーザ定義のリーダーマクロはサポートしていません．そのようなマクロを許すと，こまごました互換性のないマクロ方言であふれたコードが出来上がってしまう可能性があるからです．

Bruce Tate　既存の仮想マシンを実行プラットフォームとして選択した理由を聞かせてください．

Rich Hickey　他の言語で書かれた大規模で価値のあるコードベースが存在することは，今でこそ当然の事実ですが，Lisp が登場した当時はそうではありませんでした．とりわけ，JVM と CLR[*7]では，他の言語との相互運用性が不可欠です．
標準のマルチ言語プラットフォームによってホスト OS を抽象化して隠ぺいするという考え方そのものが，初期の Lisp が登場した頃には存在しませんでした．当時に比べると業界ははるかに大規模になり，デファクトスタンダードが登場しました．技術的には，高度なガベージコレクタや動的コンパイラ（例えば HotSpot）など，コアな技術の再利用を可能にする階層化は良いことです．ですから，Clojure は，言語がプラットフォームであるという考え方ではなく，プラットフォーム上の言語という考え方を重視しています．

Bruce Tate　確かにおっしゃるとおりです．しかし，Lisp という言語は，どうしてもう少しとっつきやすくならないのでしょうか．

Rich Hickey　理由はいろいろあります．Clojure では，括弧問題をどうにかしたいと考えました．Lisp のプログラマは，コードがデータであることの価値を知っていますが，だからと言って，括弧を嫌がる人たちを単純に切り捨てるのは間違っています．`foo(bar,`

[*7] Microsoft の共通言語ランライムで，.NET プラットフォーム向けの仮想マシン．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

baz) の代わりに (foo bar baz) と書くようにするくらいは難しいことではないと思いますが，私は，もう少し分かりやすくならないものかと考え，古い Lisp での括弧の使い方を徹底的に調べたことがあります．そうして，分かりやすくする方法をいくつか見つけたのです．古い Lisp ではすべてのものに括弧を使います．Clojure ではそうではありません．古い Lisp では，とにかく括弧が多すぎます．Clojure では，グループ化のための括弧を排除するという逆のアプローチをとっています．そのため，マクロを書く側には少し負担が大きくなっていますが，マクロを使う側から見るとかなり楽になっています．括弧を減らし，二重括弧をほとんどなくしたことで，Clojure は古い Lisp 系言語に比べて読みやすく，構造の把握や処理内容の理解がずっと容易になっています．二重括弧は Clojure のコードよりもむしろ Java のコード（例えば，忌まわしい ((AType)athing).amethod() など）でよく見られます．

1 日目で学んだこと

Clojure は JVM 上で動作する関数型言語だ．Scala や Erlang と同様，Lisp の一方言である Clojure は，関数型言語ではあるが，純粋な関数型言語ではない．具体的には，制限された形ではあるが副作用を許す．他の Lisp 方言と違って，Clojure には若干のシンタックスが追加されており，マップには中括弧を，ベクタには角括弧を使う．空白の代わりにカンマを使うこともでき，場所によっては括弧を省略できる．

Clojure の基本的なフォームの使い方を学んだ．簡単なフォームとして，ブール値，文字，数値，キーワード，文字列がある．さまざまなコレクションについても学んだ．リストとベクタはどちらも順序付きコンテナだが，ベクタはランダムアクセス用に，リストはシーケンシャルアクセス用に最適化されている．また，セット，順序なしコレクション，マップ（キーと値のペア）も使った．

関数名，パラメータリスト，関数本体，およびオプションのドキュメント文字列を指定して，名前付き関数を定義した．次に，バインディングによるデコンストラクションも使った．これにより，任意のパラメータを入力引数の任意の部分にバインドできる．この機能は，Prolog や Erlang を思い出させる．最後に，無名関数を定義し，それをマップ関数と組み合わせてリストの各要素を繰り返し処理した．

2 日目では，Clojure の再帰について見ていく．再帰は大半の関数型言語の基本構築ブロックだ．また，シーケンスと遅延評価についても解説する．これらは Clojure モデルのキモであり，コレクション上に共通の強力な抽象化階層を構築する．

この辺りで一息入れて，ここまでに学んだことを実際に試してみよう．

セルフスタディ 1 日目

Clojure は新しい言語だが，そのコミュニティは驚くほど活発で成長を続けている．本書のために調査をしたが，Clojure のコミュニティは最高の部類に入る．

探してみよう：

- Clojure シーケンスの使用例
- Clojure 関数の正式な定義

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:13:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- お使いの環境で repl を素早く起動するスクリプト

試してみよう：

- 文字列 st が n 文字より長い場合に真を返す関数 (big st n) を実装してみよ．
- col コレクションの型に応じて，:list，:map，:vector を返す関数 (collection-type col) を書け．

7.3　2 日目：マスター・ヨーダとフォース

ジェダイ・マスターであるヨーダは，フォースを使い理解するように弟子を訓練する．フォースとはあらゆる生き物の間に遍在するエネルギーである．2 日目の今日は，Clojure の基盤となる概念の説明に入る．まずシーケンスについて説明する．シーケンスは，Clojure のすべてのコレクションを統一して，Java のコレクションに結合する抽象化層だ．次に，遅延評価について説明する．遅延評価とは，シーケンスのメンバを必要になった時点で初めて計算するジャストインタイムの評価方法のことだ．そして，すべての Lisp 系言語のフォースとも言える神秘的な機能，すなわちマクロについても見ていく．

再帰：loop と recur

本書に登場した他の言語で学んだとおり，関数型言語では繰り返しではなく再帰を用いる．以下にベクタのサイズを評価する再帰プログラムを示す．

```
(defn size [v]
    (if (empty? v)
          0
          (inc (size (rest v)))))

(size [1 2 3])
```

これは難しくない．空のリストのサイズはゼロであり，その他のリストのサイズは，テールのサイズよりも 1 大きい．本書のいたるところで別の言語による同じ解法を解説した．

また，スタックが拡張されるため，再帰アルゴリズムはメモリ領域を消費し続け，最終的にはメモリ不足に陥ることも学んだ．関数型言語は，この制限を末尾再帰最適化によって回避している．Clojure では JVM の制約があるため，末尾再帰最適化を暗黙ではサポートしていない．末尾再帰最適化を行うには，loop と recur を用いて明示的に繰り返す必要がある．ループは let 文と考えてもらえればよい．

```
(loop [x x-initial-value, y y-initial-value] (do-something-with x y))
```

ベクタを初期値として与えると，loop は偶数位置にある変数を奇数位置にある値にバインドする．実際，recur を指定しなければ，loop は let とまったく同じように動作する．

```
user=> (loop [x 1] x)
1
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

関数 recur は再度ループを呼び出すが，今度は新しい値を渡す．
size 関数を recur を使うようにリファクタリングしてみよう．

```
(defn size [v]
        (loop [l v, c 0]
        (if (empty? l)
                c
                (recur (rest l) (inc c)))))
```

この新しいバージョンの size では，末尾再帰最適化するための loop と recur を用いている．recur は値を返すわけではないので，計算結果はアキュムレータと呼ばれる変数に保存することになる．この例では，c にカウントを保持している．

このバージョンは末尾最適化呼び出しのように動作するが，コードがごちゃごちゃして読みづらい．JVM は，ときとして諸刃の剣である．Java のコミュニティの恩恵を受けたいなら，この末尾再帰問題に対処する必要がある．しかし，この関数は，いくつかの基本的なコレクション API に組み込まれているため，直接 recur を使う必要があるケースはあまり多くない．また Clojure では，遅延シーケンスなど，再帰に代わる優れた手法をいくつか用意している．これについては本章で後述する．

2 日目で取り上げるトピックの問題点がクリアになった．これで，いつでも楽しいトピックに移ることができる．まずはシーケンスからだ．シーケンスについて説明する過程で，Clojure を特別な言語にしているいくつかの機能が登場する．

シーケンス

シーケンスは，Clojure という言語に登場するさまざまなコンテナを実装に依存しない形で抽象化したものだ．シーケンスは，Clojure のすべてのコレクション（セット，マップ，ベクタなど），文字列，およびファイルシステム構造までもラッピングする．また，Java コンテナ（Java のコレクション，配列，文字列など）のための共通の抽象化も提供する．一般に，first，rest，cons の各関数をサポートしているものは，シーケンスでラッピングできる．

先ほどベクタを操作していたとき，次のようにコンソールにリストが返されたことがあった．

```
user=> [1 2 3]
[1 2 3]
user=> (rest [1 2 3])
(2 3)
```

最初はベクタから始めた．得られた結果はリストではない．実は，repl はシーケンスを返したのである．つまり，Clojure では，すべてのコレクションを総称的に扱うことができる．それでは，共通のシーケンスライブラリについて見ていこう．このライブラリは内容が豊富で強力なため，1 つの章の 1 つのセクションではとてもすべてを解説することはできないが，何ができるのかについて感じだけでもつかんでもらえればと思う．ここでは，シーケンスを変更，テスト，作成する関数が登場するが，説明はごく簡単に済ませることにする．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

テスト

シーケンスをテストするときは，述語と呼ばれる関数を使う．述語はシーケンスとテスト関数を引数にとり，ブール値を返す．例えば，every?は，テスト関数がシーケンス内のすべての項目について真ならば true を返す．

```
user=> (every? number? [1 2 3 :four])
false
```

項目の 1 つが数値ではないので false が返された．some は，テストが，シーケンス内のいずれかの項目について真ならば true を返す[*8]．

```
user=> (some nil? [1 2 nil])
true
```

項目の 1 つが nil なので true が返された．not-every?と not-any?は，上記 2 つの述語を否定したものだ．

```
user=> (not-every? odd? [1 3 5])
false
user=> (not-any? number? [:one :two :three])
true
```

期待どおりに動作する．次に，シーケンスを変更する関数の説明に移ろう．

シーケンスの変更

シーケンスライブラリには，シーケンスをさまざまな方法で変換する関数が多数用意されている．filter については既に見た．例えば，長さが 4 を超える単語だけを取り出すには，次のように書く．

```
user=> (def words ["luke" "chewie" "han" "lando"])
#'user/words
user=> (filter  (fn [word] (> (count word) 4)) words)
("chewie" "lando")
```

map についても既に見た．コレクションのすべての項に対して関数を呼び出し，結果を返すのが map だ．例えば，ベクタ内のすべての項の 2 乗のシーケンスを作るには次のように書く．

```
user=> (map (fn [x] (* x x)) [1 1 2 3 5])
(1 1 4 9 25)
```

リスト内包表記は，Erlang と Scala で見たように，マップとフィルタを組み合わせたものだ．リスト内包表記は，複数のリストのすべての組み合わせに対してフィルタを適用することによって，複数のリストとフィルタを組み合わせることを思い出してほしい．まず簡単なケースを見てみよう．次に示す 2 つのリスト colors と toys がある．

```
user=> (def colors ["red" "blue"])
#'user/colors
```

[*8] 厳密には，some は，nil でも false でもない最初の値を返す．例えば (some first [[] [1]]) は 1 を返す．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> (def toys ["block" "car"])
#'user/toys
```

リスト内包表記を用いてすべての色に対して関数を適用する．これは map の動作方法と同じだ．

```
user=> (for [x colors] (str "I like " x))
("I like red" "I like blue")
```

[x colors] は x を色リストの 1 要素にバインドする．(str "I like " x) は，colors の各 x に対して適用される任意の関数だ．複数のリストにバインドすると，さらに面白くなる．

```
user=> (for [x colors, y toys] (str "I like " x " " y "s"))
("I like red blocks" "I like red cars"
 "I like blue blocks" "I like blue cars")
```

2 つのリストから考えられるすべての組み合わせが作成される．バインディングに :when キーワードを使ってフィルタを適用することもできる．

```
user=> (defn small-word? [w] (< (count w) 4))
#'user/small-word?
user=> (for [x colors, y toys, :when (small-word? y)]
           (str "I like " x " " y "s"))
("I like red cars" "I like blue cars")
```

small-word?という名前のフィルタを作成した．長さが 4 文字に満たないすべての単語は小さい単語である．:when (small-word? y) によって，small-word?フィルタを y に適用する．これにより，x が colors のメンバ，y が toys のメンバ，y のサイズが 4 文字に満たないようなすべての (x, y) の組み合わせが得られる．コードは高密度だが表現力に富んでいる．これは理想的な組み合わせだ．では次に進もう．

foldl，foldleft，および inject については，Erlang，Scala，および Ruby で既に説明した．Lisp でこれらに相当するのは reduce だ．例えば，合計および階乗を計算するには次のようにする．

```
user=> (reduce + [1 2 3 4])
10
user=> (reduce * [1 2 3 4 5])
120
```

リストをソートすることもできる．

```
user=> (sort [3 1 2 4])
(1 2 3 4)
```

関数の実行結果をソートする．

```
user=> (defn abs [x] (if (< x 0) (- x) x))
#'user/abs
user=> (sort-by abs [-1 -4 3 2])
(-1 2 3 -4)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

絶対値を計算する関数 abs を定義してから，その関数をソートの引数として使う．ここまで，Clojure で最も重要なシーケンス変換の例を紹介した．次は，シーケンスを作成する関数の説明に移る．遅延を好む「怠け者」になって理解してほしい．

遅延評価

数学では，無限数列のほうが容易に記述できることが多い．関数型言語でも，そうした簡単な記述方法があればよいのにと思われがちだが，実際には無限数列を計算することはできない．そこで登場するのが遅延評価だ．遅延評価を用いることによって，Clojure のシーケンスライブラリは，そのシーケンスが実際に消費（使用）されるときになって初めて値を計算する．実際，ほとんどのシーケンスは遅延評価される．最初にいくつか有限数列を作成し，その後，遅延シーケンスの説明に入ることにしよう．

範囲を使った有限数列

Ruby と違って，Clojure は範囲を関数として実装している．範囲はシーケンスを作成する．

```
user=> (range 1 10)
(1 2 3 4 5 6 7 8 9)
```

上限は含まれない点に注意してほしい．したがって生成されたシーケンスには 10 は含まれない．増分には任意の値を指定できる．

```
user=> (range 1 10 3)
(1 4 7)
```

増分を指定しない場合は下限を省略できる．

```
user=> (range 10)
(0 1 2 3 4 5 6 7 8 9)
```

デフォルトの下限は 0 だ．範囲を用いて作成されたシーケンスは有限だ．上限なしのシーケンスを作成するにはどうすればよいだろうか．次のセクションで考えてみよう．

無限シーケンスと take

最も基本的な無限数列，すなわち，同じ要素を繰り返す無限数列から始めよう．例えば，(repeat 1) と指定する．これを repl で試してみると，プロセスを殺すまで 1 が繰り返し表示される．当然，その一部分を何らかの方法で取り出す必要がある．それを行う関数が take だ．

```
user=> (take 3 (repeat "Use the Force, Luke"))
("Use the Force, Luke" "Use the Force, Luke" "Use the Force, Luke")
```

この例では，"Use the Force, Luke"という文字列を繰り返す無限数列を作成し，その最初の 3 項を取り出している．cycle を使ってリスト内の要素を繰り返すこともできる．

```
user=> (take 5 (cycle [:lather :rinse :repeat]))
(:lather :rinse :repeat :lather :rinse)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ベクタ [:lather :rinse :repeat] を cycle で回して出来たシーケンスの最初の 5 要素を取り出している．問題ない．シーケンスの最初の数要素を落とすこともできる．

```
user=> (take 5 (drop 2 (cycle [:lather :rinse :repeat])))
(:repeat :lather :rinse :repeat :lather)
```

内側から外側に向かって順に，前と同様 cycle（循環）シーケンスを生成し，最初の 2 項を落とし，その後の 5 項を取り出している．しかし，内側から処理する必要はない．新しく導入された「左から右演算子」（->>）を使えば，各関数を結果に適用できる．

```
user=> (->> [:lather :rinse :repeat] (cycle) (drop 2) (take 5))
(:repeat :lather :rinse :repeat :lather)
```

やはり，ベクタを受け取り，cycle でシーケンスを生成し，最初の 2 項を落とし，その後の 5 項を取り出している．このように，場合によっては左から右にコーディングしたほうが読みやすい．単語の間に区切り文字を挟みたい場合はどうすればよいだろう？ その場合は interpose を用いる．

```
user=> (take 5 (interpose :and (cycle [:lather :rinse :repeat])))
(:lather :and :rinse :and :repeat)
```

:and というキーワードを受け取って，無限シーケンスのすべての要素の間に挿入している．interpose 関数は，Ruby の join を一般化したものと思えばよい．別のシーケンスのメンバを順に間に挿入していくには（つまり，2 つのシーケンスの各項を交互に並べていくには）どうすればよいだろうか？ それには interleave を用いる．

```
user=> (take 20 (interleave (cycle (range 2)) (cycle (range 3))))
(0 0 1 1 0 2 1 0 0 1 1 2 0 0 1 1 0 2 1 0)
```

2 つの無限シーケンス (cycle (range 2)) と (cycle (range 3)) を交互に配置（インターリーブ）し，その最初の 20 項を取り出している．生成されたシーケンスを見れば分かるように，偶数番目が (0 1 0 1 0 1 0 1 0 1)，奇数番目が (0 1 2 0 1 2 0 1 2 0) になっている．美しい．

iterate 関数を使うと，別の方法でシーケンスを生成できる．次の例を見てほしい．

```
user=> (take 5 (iterate inc 1))
(1 2 3 4 5)
user=> (take 5 (iterate dec 0))
(0 -1 -2 -3 -4)
```

iterate は関数と開始値を受け取り，その関数を開始値に繰り返し適用する．上の 2 つの例ではそれぞれ，inc と dec を繰り返し呼び出している．

次に，フィボナッチ数列の連続するペアを計算する例を示す．フィボナッチ数列の各項は直前の 2 項の和であることを思い出してほしい．1 つのペア [a b] が与えられると，次のペアは [b，a + b] で生成できる．したがって，直前の値が与えられたとき，現在のペアを生成する無名関数は次のようになる．

```
user=> (defn fib-pair [[a b]]  [b (+ a b)])
#'user/fib-pair
user=> (fib-pair [3 5])
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
[5 8]
```

次に，iterate を用いて，無限シーケンスを生成する．まだ実行しないように．

```
(iterate fib-pair [1 1])
```

map を用いて各ペアの最初の要素を取り出す．

```
(map
first
(iterate fib-pair [1 1]))
```

これもまだ無限シーケンスだ．そこで，最初の 5 項を取り出す．

```
user=> (take 5
       (map
       first
       (iterate fib-pair [1 1])))
(1 1 2 3 5)
```

500 番目の項を取り出すこともできる．

```
user=> (nth (map first (iterate fib-pair [1 1])) 500)
(225... 途中省略...626)
```

パフォーマンスは素晴らしい．遅延シーケンスを使うと，フィボナッチ数列のようなよくある再帰問題を記述できる．次のように，階乗計算も記述できる．

```
user=> (defn factorial [n] (apply * (take n (iterate inc 1))))
#'user/factorial
user=> (factorial 5)
120
```

無限シーケンス (iterate inc 1) から n 個の要素を取り出した後，apply *でそれらの各要素を掛け合わせている．この解法はいたってシンプルだ．以上，遅延評価について，かなりページを割いて説明した．次に，Clojure の新しい関数 defrecord と protocol について説明しよう．

defrecord と protocol

これまで，Java との高レベルの統合について話してきたが，Clojure に JVM の片鱗が顔をのぞかせるところはあまり登場していない．JVM とは，結局のところ，型とインタフェースだ（Java プログラマではない方のために：型とは Java クラス，インタフェースとは実装されていない Java クラスと考えてよい）．JVM との統合性を高めるため，Clojure の最初の実装には，かなりの量の Java コードが含まれていた．

Clojure の処理速度が向上し，効率的な JVM 言語であることが立証されるようになると，Clojure のもっと多くの部分を Clojure で実装したいという要望が強くなった．そのために，Clojure の開発者たちは，実装に依存しない抽象的なプログラミングによって，プラットフォームに相応な速さで動作するオープンな拡張ライブラリを構築する方法を必要としていた．そこで登場したのが，型を定義する defrecord と，型によって関数をグループ化する protocol だ．Clojure 側から見ると，オブジェクト指向の最も良いとこ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ろは型とプロトコル（インタフェースなど）であり，最も悪いところは実装の継承だ．Clojure の defrecord と protocol はオブジェクト指向の良いところを残し，それ以外の部分を切り捨てる．

これらの機能は重要だが，本書の執筆時点では，まだ開発中である．そこで，Stuart Halloway 氏（Relevance の共同創設者で『プログラミング Clojure』[Hal09] の著者）から実地に学んで，実用的な実装を概説してみる．ここでは JVM 上で動作する別の関数型言語である Scala に戻って，Clojure で Compass プログラムを書き直してみる．それでは始めよう．

まずプロトコルを定義する．Clojure の protocol は契約のようなものだ．このプロトコルの型は，特定の関数，フィールド，および引数をサポートする．以下に，Compass を記述する protocol を示す．

▶ code/clojure/compass.clj

```
(defprotocol Compass
  (direction [c])
  (left [c])
  (right [c]))
```

このプロトコルは，Compass という抽象概念を定義し，Compass でサポートする必要がある関数（direction，left，right）をその引数の数と一緒に列挙する．これで，defrecord を用いてこのプロトコルをいつでも実装できる．次に，4 つの方角を定義する必要がある．

▶ code/clojure/compass.clj

```
(def directions [:north :east :south :west])
```

体の向きを制御する関数が必要だ．基本方位は整数 0, 1, 2, 3 であり，それぞれ:north，:east，:south，:west に対応することを思い出してほしい．基本方位に 1 を足すたびに，コンパスは 90 度右に回転する．(base+amount)/4（正確には (base+amount)/方位の数）の余りを計算して，:west の次が:north に正しく折り返す（循環する）ようにする．

▶ code/clojure/compass.clj

```
(defn turn
  [base amount]
  (rem (+ base amount) (count directions)))
```

これで体の向きは，こちらの期待どおりに変わる．コンパス・ファイルをロードして，turn 関数を使ってみる．

```
user=> (turn 1 1)
2
user=> (turn 3 1)
0
user=> (turn 2 3)
1
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

文章にすると，:east から 1 回右に向くと:south になり，:west から 1 回右に向くと:north になり，:south から 3 回右に向くと:east になる．

ではプロトコルの実装に移ろう．実装には defrecord を使用し，段階に分けて行う．まず defrecord を使って，次のようにプロトコルを実装することを宣言する．

▶ code/clojure/compass.clj

```
(defrecord SimpleCompass [bearing]
  Compass
```

SimpleCompass という名前の新しいレコードを定義している．このレコードには 1 つのフィールド bearing が存在する．次に，Compass プロトコルを実装する．最初は direction 関数だ．

▶ code/clojure/compass.clj

```
(direction [_] (directions bearing))
```

direction 関数は，インデックスが bearing の位置の directions の要素を検索する．例えば，(directions 3) は:west を返す．各引数リストは，インスタンスへの参照（例えば，Ruby の self や Java の this など）を保持しているが，ここではそれは使わないため，引数リストの位置には_を指定している．次は left 関数と right 関数だ．

▶ code/clojure/compass.clj

```
(left [_] (SimpleCompass. (turn bearing 3)))
(right [_] (SimpleCompass. (turn bearing 1)))
```

Clojure では，変更不能な値を使うことを思い出してほしい．つまり，向きを変えると，既存のコンパスをその場で変更するのではなく，新しい変更されたコンパスが返される．left と right はどちらも，これまで見たことのないシンタックスを使っている．(SomeType. arg) は，SimpleCompass のコンストラクタを起動し，その際，arg を第 1 パラメータにバインドせよ，という意味だ．repl に (String. "new string") と入力すると新しい文字列"new string"が返されることが確認できる．

left および right 関数は簡単だった．どちらも前述の turn 関数を用いて，適切なベアリングを持ち適切に設定された新しいコンパスを返す．right は右に 90 度 1 回回転し，left は右に 90 度 3 回回転する．ここまでで，Compass プロトコルを実装する型 SimpleCompass を定義した．文字列表現を返す関数が必要だが， toString は java.lang.Object のメソッドだ．これを SimpleCompass 型に追加するのは次のように簡単だ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/clojure/compass.clj

```
Object
(toString [this] (str "[" (direction this) "]")))
```

次に，Object プロトコルの一部を toString メソッドで実装して，SimpleCompass [:north] という形式の文字列を返す．

これで型は完成だ．新しいコンパスを作成してみる．

```
user=> (def c (SimpleCompass. 0))
#'user/c
```

向きを変えると新しいコンパスが返される．

```
user=> (left c) ; returns a new compass
#:SimpleCompass{:bearing 3}

user=> c ; original compass unchanged
#:SimpleCompass{:bearing 0}
```

古いコンパスは変更されていない点に注意してほしい．JVM の型を定義しているため，すべてのフィールドは Java フィールドとしてアクセスできるが，型内のフィールドには Clojure マップと同様にキーワードでアクセスすることもできる．

```
user=> (:bearing c)
0
```

これらの型はマップとして動作するため，新しい型のプロトタイプをマップとして作成しておき，設計が安定するまでは，それをそのつど型に変換して使うこともできる．スタブやモックとしてのテストでは型をマップで置換することもできる．

- 型は他の Clojure の並行性構文と相性が良い．3 日目で，リレーショナルデータベースのようにトランザクション整合性が維持されるように，Clojure オブジェクトの変更可能な参照を作成する方法を学ぶ．
- protocol を実装したわけだが，しかしプログラムを書く際にいつでも protocol を使わなければいけないわけではない．作成するのは JVM の型なので，Java のクラスやインタフェースと同じように扱うことができる．

defrecord と protocol により，Clojure は，Java のコードを書かずに，JVM ネイティブのコードを生成できる機能を提供している．このようにして書かれたコードは，Java のクラスやインタフェースを含む，JVM 上の他の型と相互にやり取りできる．Clojure の型を基に Java のクラスを作成することもできる．もちろん Java との相互運用性の話はこれだけではないが，defrecord と protocol は極めて重要な部分だ．Java を拡張する方法を学んだところで，Clojure 自身をマクロで拡張する方法を学ぼう．

マクロ

このセクションでは Io の章を参照する．3.3「メッセージ」(p. 58)では，Ruby の unless を実装した．形式は (unless test form1) だ．この関数は test が false のとき form1

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:14:53 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell―［2011］オーム社

を実行する．これをそのままコーディングしてもうまくいかない．というのは，各パラメータが実行されてしまうからだ．

```
user=> ; Broken unless
user=> (defn unless [test body] (if (not test) body))
#'user/unless
user=> (unless true (println "Danger, danger Will Robinson"))
Danger, danger Will Robinson
nil
```

この問題は Io の章でも説明した．大半の言語では，まずパラメータを実行してから，その結果を呼び出しスタックに積む．しかし，この例では，条件が偽のときだけブロックを評価するようにしたい．Io では，unless メッセージの実行を遅延することでこの問題を回避していた．Lisp では，マクロを使う．(unless test body) と入力されたら，(if (not test) body) という文に変換されるようにしたい．そこで登場するのがマクロだ．

Clojure のプログラムは 2 段階に分けて実行される．マクロ展開機能によって，Clojure プログラム内のすべてのマクロが展開された形式に変換される．マクロ展開によって実行される処理の内容を確認するには，macroexpand コマンドを実行する．実はこれまでに既にマクロを使っている．リーダーマクロというマクロだ．セミコロン（;）はコメント，シングルクオート（'）は quote，ハッシュ記号（#）は無名関数だ．時期尚早な実行を抑制するには，展開する式の前にクオートを付ける．

```
user=> (macroexpand ''something-we-do-not-want-interpreted)
(quote something-we-do-not-want-interpreted)
user=> (macroexpand '#(count %))
(fn* [p1__97] (count p1__97))
```

これらがマクロだ．一般に，マクロ展開を使うと，コードをリストのように扱うことができる．関数をすぐに実行したくないときはシングルクオートを前に付ければよい．クオートされた引数はそのままで一切処理されない．正しい unless は次のようになる．

```
user=> (defmacro unless [test body]
         (list 'if (list 'not test) body))
#'user/unless
```

ここで Clojure は test と body を評価せずにそのまま使っている点に注目してほしい．ただし，if と not はクオートしたうえで，リスト内に含める必要がある．Clojure が実行する形式でコードのリストを作成したわけだ．このコードを macroexpand コマンドで展開してみる．

```
user=> (macroexpand '(unless condition body))
(if (not condition) body)
```

実行してみよう．

```
user=> (unless true (println "No more danger, Will."))
nil
user=> (unless false (println "Now, THIS is The FORCE."))
Now, THIS is The FORCE.
nil
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

実は，ここで行ったことは，言語の基本定義の変更である．言語設計者にキーワードの追加を要求することなく，自分で制御構文を追加したわけだ．マクロ展開はおそらく，Lisp で最も強力な機能であり，同等の機能を提供している言語はほとんどない．秘密は，データを単なる文字列ではなくコードとして表現できる点にある．このコードは既に高階データ構造になっている．

2 日目のまとめをしよう．実に盛りだくさんだった．演習の前に一息入れることにしよう．

2 日目で学んだこと

2 日目も盛りだくさんだった．膨れる一方のコーディング技法の袋にまた，たくさんの抽象化を追加した．復習しておこう．

まず，再帰の使い方について学んだ．JVM は末尾再帰最適化をサポートしていないため，`loop` と `recur` を使う必要があった．このループ構文によって，通常は再帰関数によって実装するアルゴリズムを多数実装できるが，シンタックスは再帰呼び出しから遠ざかってしまう．

シーケンスも使った．Clojure では，シーケンスによって，あらゆるコレクションに対するアクセスをカプセル化する．共通ライブラリを使うと，共通の戦略を適用してコレクションを処理できる．シーケンスの変更，変換，検索を行うさまざまな関数を紹介した．高階関数によって，シーケンスライブラリの処理能力とシンプルさを強化する例についても学んだ．

遅延シーケンスを使うと，シーケンスにさらに強力なレイヤを追加することができる．遅延シーケンスは，アルゴリズムを簡素化する．また遅延実行を実現することで，パフォーマンスが大幅に向上し，疎結合が推進される．

次に，型の実装についてページを割いて説明した．`defrecord` と `protocols` を用いることで，JVM に完全に準拠した型を実装できる．

最後に，マクロを使って言語自体に機能を追加した．Clojure は，マクロ展開と呼ばれる段階を経て，コードを実装または解釈する．マクロ展開内で `if` 関数を用いることにより，`unless` を実装した．

2 日目は理解しなければならないことが多い．少し時間を割いて，次の演習に取り組んでみよう．

セルフスタディ 2 日目

2 日目の今日は，Clojure の最も高度で強力な要素を詰め込み気味に説明した．少し時間をとって，紹介した機能を理解するようにしてほしい．

探してみよう：

- Clojure でよく使われるマクロの実装
- 独自の遅延シーケンスの定義例
- `defrecord` および `protocol` 機能の最新状況（これらの機能は，本書の執筆時点ではまだ開発中であった）

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

試してみよう：

- マクロを使って `else` 付きの `unless` 構文を実装してみよ．
- `defrecord` を使って，プロトコルを実装する型を書け．

7.4　3日目：悪を見逃すな

『スターウォーズ』では，ヨーダが最初にダース・ベイダーに潜む悪を看破する．Rich Hickey は，Clojure によって，並行オブジェクト指向システムの開発者を悩ませる重要な問題を特定した．これまでもたびたび述べてきたが，可変状態は，オブジェクト指向プログラムの中核部分に潜む悪である．これまでにも可変状態に対処するさまざまな方法を紹介してきた．Io と Scala では，アクターベースのモデルを用いて，プログラマが可変状態を使わずにこれらの問題を解決できるようにする変更不能な構文を用意した．Erlang では，軽量プロセスを用いたアクターと，効果的なモニタリングと通信を可能にする仮想マシンによって，比類のない信頼性を実現した．Clojure の並行性に対するアプローチは，他の言語とは異なる．Clojure では，ソフトウェアトランザクショナルメモリ（STM）を使う．このセクションでは，STM，およびスレッド対応のアプリケーション間で状態を共有するためのツールについて説明する．

参照とトランザクションメモリ

データベースはトランザクションによってデータの整合性を保証する．最近のデータベースは少なくとも 2 種類の並行性制御機能を使用している．ロックは，2 つの競合するトランザクションが同時に同じ行にアクセスするのを防ぐ．バージョニングは，各トランザクションがそれぞれ競合データのコピーを保持できるように複数のバージョンを用意する．別のトランザクションと干渉するトランザクションが存在する場合，データベースエンジンは単にそのトランザクションを終了する．

Java などの言語では，あるスレッドが，自身のリソースを破壊する可能性のある競合スレッドからの保護を実現するのに，ロックを用いる．ロックは基本的に，並行制御のための作業をプログラマに負担させるものだ．しかしこの負担は耐え難いものになっている．最近は特にその傾向が強い．

Clojure を始めとするいくつかの言語ではソフトウェアトランザクショナルメモリ（STM）が利用可能だ．この戦略は複数のバージョンを用いて一貫性と整合性を維持するものだ．Scala，Ruby，Io と違って，Clojure で参照状態を変更する場合は，トランザクションのスコープ内で行う必要がある．その動作原理を説明しよう．

参照

Clojure では，*ref*（reference の略称）は，ラップされたデータ片である．すべてのアクセスは，指定されたルールに準拠しなければならない．この場合のルールとは STM をサポートすることだ．トランザクションの外で参照を変更することはできない．その動作方法を確認するために，参照を作成してみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
user=> (ref "Attack of the Clones")
#<Ref@ffdadcd: "Attack of the Clones">
```

これではあまり面白くない．次のように，参照を値に代入することにしよう．

```
user=> (def movie (ref "Star Wars"))
#'user/movie
```

参照を取得するには次のようにする．

```
user=> movie
#<Ref@579d75ee: "Star Wars">
```

しかし，我々が欲しいのは参照先の値である．これを確認するには deref を使う．

```
user=> (deref movie)
"Star Wars"
```

deref の略記法もある．

```
user=> @movie
"Star Wars"
```

随分良くなった．これで，参照内の値に簡単にアクセスできる．ここまでは，参照の状態を変更しようとはしていない．変更操作を試してみよう．Clojure では，値を変更する関数を送信する．ref がデリファレンスされ，関数の第 1 引数として渡される．

```
user=> (alter movie str ": The Empire Strikes Back")
java.lang.IllegalStateException: No transaction running (NO_SOURCE_FILE:0)
```

見てのとおり，状態はトランザクション内でしか変更できない．トランザクションを開始するには dosync 関数を用いる．参照を変更するには，次のように，特定の変換関数を用いるのが望ましい．

```
user=> (dosync (alter movie str ": The Empire Strikes Back"))
"Star Wars: The Empire Strikes Back"
```

ref-set で初期値を設定することもできる．

```
user=> (dosync (ref-set movie "Star Wars: The Revenge of the Sith"))
"Star Wars: The Revenge of the Sith"
```

参照が変更されていることを確認してみる．

```
user=> @movie
"Star Wars: The Revenge of the Sith"
```

期待どおりだ．参照が変更されている．このような方法で変更可能な変数を変更するのは手間だが，Clojure ではこのちょっとしたポリシーを守らせることで後の苦労を大幅に軽減している．このポリシーに従って動作するプログラムは，競合状態とデッドロックに関しては，完全に正しく実行されることが分かっている．我々の書くコードの大半は関数型パラダイムに基づくので，STM を使うのは，可変性の利点が最大に発揮される問題のためにとっておこう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

アトムを使う

他のアクティビティとの連係のない単一の参照についてスレッド安全性を保証したい場合は，アトムを使う．これらのデータ要素は，トランザクションのコンテキスト外での変更を許す．Clojure のアトムは，参照と同様，カプセル化された状態データだ．試してみよう．まず，アトムを作成する．

```
user=> (atom "Split at your own risk.")
#<Atom@53f64158: "Split at your own risk.">
```

アトムをバインドする．

```
user=> (def danger (atom "Split at your own risk."))
#'user/danger
user=> danger
#<Atom@3a56860b: "Split at your own risk.">
user=> @danger
"Split at your own risk."
```

danger を新しい値にバインドするには，reset!を使う．

```
user=> (reset! danger "Split with impunity")
"Split with impunity"
user=> danger
#<Atom@455fc40c: "Split with impunity">
user=> @danger
"Split with impunity"
```

reset!はアトム全体を置換するが，アトムを変換する関数を用意するほうが使い方として望ましい．大きなベクタを変更する場合は，次のように swap!を使ってその場でアトムを変更できる．

```
user=> (def top-sellers (atom []))
#'user/top-sellers
user=> (swap! top-sellers conj {:title "Seven Languages", :author "Tate"})
[{:title "Seven Languages in Seven Weeks", :author "Tate"}]
user=> (swap! top-sellers conj {:title "Programming Clojure" :author "Halloway"})
[{:title "Seven Languages in Seven Weeks", :author "Tate"}
 {:title "Programming Clojure", :author "Halloway"}]
```

参照の場合と同様，値を一度作成したら，その値を swap!で変更したいだろう．実用的な例を見てみよう．

アトム・キャッシュの生成

ここまで，参照とアトムの両方について見てきた．Haskell を使う際にも全般的な方針は同じである．つまり，状態データをパッケージでラップし，後で関数を用いて変更できるようにする．参照はトランザクションを必要とするが，アトムにはトランザクションは不要である．簡単なアトム・キャッシュを作成してみよう．これは，アトムには格好の問題だ．ここでは，単純にハッシュを用いて，名前を値に関連付ける．この例は，Clojure のトレーニングとコンサルティングを提供しているコンサルタント会社 Relevance[*9]の

[*9] http://www.thinkrelevance.com

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Stuart Halloway 氏の好意により掲載する．

まずキャッシュを作成する必要がある．その後，キャッシュへの要素の追加，およびキャッシュからの要素の削除を実行する関数が必要になる．

▶ code/clojure/atomcache.clj

```
(defn create
  []
  (atom {}))
```

ここではアトムを作成しているだけだ．アトムのバインドはこのクラスのクライアントに行わせる．次に，キャッシュのキーを取得できなければならない．

▶ code/clojure/atomcache.clj

```
(defn get
  [cache key]
  (@cache key))
```

キャッシュとキーを引数として受け取る．キャッシュはアトムなので，これをデリファレンスして，キーに対応する項目を返す．最後に，項目をキャッシュに格納する必要がある．

▶ code/clojure/atomcache.clj

```
(defn put
  ([cache value-map]
     (swap! cache merge value-map))
  ([cache key value]
     (swap! cache assoc key value)))
```

ここでは，put という関数の 2 つの異なるバージョンを定義している．最初のバージョンでは，merge を用いてマップ内のすべての対応付けをキャッシュに追加する．2 番目のバージョンでは，assoc を用いてキーと値を追加する．以下にキャッシュの使用例を示す．キャッシュに項目を追加し，それを返している．

▶ code/clojure/atomcache.clj

```
(def ac (create))
(put ac :quote "I'm your father, Luke.")
(println (str "Cached item: " (get ac :quote)))
```

出力結果は次のようになる．

```
Cached item: I'm your father, Luke.
```

アトムと参照は，可変状態を同期的に処理する簡単で安全な方法だ．以降のセクションでは，可変状態を非同期に処理する 2 つの例を見てみよう．

エージェントを使う

アトム同様，エージェントは，ラップされたデータ片だ．Io のフューチャ同様，デリファレンスされたエージェントの状態は，値が使用可能になるまでブロックされる．ユー

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ザは関数を使ってデータを非同期に変更できる．更新は別スレッドで行われる．2つ以上の関数が同時にエージェントの状態を変更することはできない．

試してみよう．

```
user=> (defn twice [x] (* 2 x))
#'user/twice
```

次に，初期値1を持つエージェントtribblesを定義する．

```
user=> (def tribbles (agent 1))
#'user/tribbles
```

これでエージェントに値を送信することでtribblesを変更できる．

```
user=> (send tribbles twice)
#<Agent@554d7745: 1>
```

この関数は別スレッドで実行される．エージェントの値を取得してみよう．

```
user=> @tribbles
2
```

参照，エージェント，またはアトムから値を読み取っても，ロックやブロック（処理待ち）が発生することはない．読み取りは高速でしかるべきで，正しい抽象化を行えばそれは実現可能だ．この関数を使うと，次のように，各エージェントから読み取る値が異なることを確認できる．

```
user=> (defn slow-twice [x]
         (do
           (Thread/sleep 5000)
           (* 2 x)))
#'user/slow-twice
user=> @tribbles
2
user=> (send tribbles slow-twice)
#<Agent@554d7745: 16>
user=> @tribbles
2
user=> ; 5秒後に以下を実行する
user=> @tribbles
4
```

シンタックスに関してはあまり悩まなくてよい．(Thread/sleep 5000)は単純にJavaのThreadクラスのsleepメソッドを呼び出しているだけだ．ここでは，エージェントの値に注目してほしい．5秒かかる遅いバージョンのtwiceを定義した．これは，時間の経過に伴う@tribblesの値の変化を確認するのに十分な時間である．

さて，これでtribblesのある時点での値を取得できる．ただし，自分自身のスレッドから行った最新の変更内容は取得できないかもしれない．自分自身のスレッドから行った最新の値を確実に取得するには，(await tribbles)または(await-for timeout tribbles)を呼ぶ（timeoutはミリ秒単位のタイムアウト値）．awaitおよびawait-forは，自分のスレッドから行ったアクションがディスパッチされると，ブロック解除される点を覚えておいてほしい．他のスレッドが自分のスレッドに依頼してきた内容に関して

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

は関知していない．つまり，何かの最新値を取得したいと考えているなら，その時点で既に失敗している．Clojure のツールはスナップショットを使う必要があるが，スナップショットの値は一瞬を切り取ったものであり，その時点で既に古いデータになっている可能性がある．これはまさに，高速な並行制御の実現のためにデータベースのバージョニングが行っていることである．

フューチャ

Java では，特定のタスクを直接解決するためにスレッドを開始する．もちろん，Java との統合化により，この方法でスレッドを開始することはできるが，たいていはもっと良い方法がある．例えばカプセル化された状態に対する複雑な計算を処理するスレッドを作成したいとする．これにはエージェントが使えるだろう．あるいは，値の計算を開始したいが，結果が返されるまで待ちたくない場合はどうだろうか．Io の場合と同様にフューチャが使える．具体的に見ていこう．

まずフューチャを作成しよう．フューチャは即座に参照を返す．

```
user=> (def finer-things (future (Thread/sleep 5000) "take time"))
#'user/finer-things
user=> @finer-things
"take time"
```

タイピングの速さにもよるが，結果が返ってくるまで待たされた人もいるだろう．フューチャは 1 つ以上の式からなるボディ部を受け取り，最後の式の値を返す．フューチャは別スレッドで開始される．フューチャをデリファレンスすると，その値が使用可能になるまでブロックされる．

つまりフューチャは，計算が完了する前に非同期に復帰できる並行性構文である．フューチャを使えば，処理に時間のかかる複数の関数を並列に実行できる．

説明しきれなかった点

Clojure は Lisp である．Lisp というだけでも非常に表現力の豊かな言語だが，Clojure は JVM を基盤としている．JVM には，10 年以上におよぶ開発の蓄積がある．Clojure には，新しい強力な概念も取り込まれている．1 冊の本の 1 つの章で Clojure をすべて説明するのは不可能だ．この章では説明しなかったが，知っておくべきことを以下にいくつかまとめておく．

メタデータ

メタデータを型に関連付けるのは，場合によっては良い考えだ．Clojure では，シンボルとコレクションの両方にメタデータを添付し，それらにアクセスできる．`(with-meta value metadata)` という構文によって，`metadata`（通常はマップとして実装）に新しい `value` を関連付けることができる．

Java との統合

Clojure は Java との統合性が極めて高い．Java との統合性については，本文でもごく簡単に説明し，JVM に基づく型も作成した．既存の Java ライブラリはまったく使わなかっ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

た．Java互換性フォームについても詳しくは説明しなかった．例えば（.toUpperCase "Fred"）と書くと，文字列"Fred"に対して.toUpperCaseメンバ関数が呼び出される．

マルチメソッド

オブジェクト指向言語は，振る舞いとデータを編成するための1つのスタイルを実現する．Clojureでは，マルチメソッドによってユーザが独自のコード編成を構築できる．関数ライブラリに型を関連付けることができる．また，マルチメソッドを用いて，型，メタデータ，引数，さらには属性に基づいてメソッドのディスパッチを行うことでポリモーフィズムを実装できる．この概念は強力で極めて柔軟だ．例えば，Javaスタイルの継承，プロトタイプの継承，あるいは全く異なる何かを実装することも可能である．

スレッドの状態

Clojureでは，アトム，参照，エージェントなどにより，さまざまな並行モデルを提供している．場合によっては，スレッドインスタンスごとにデータを格納する必要がある．Clojureではこれを vars によって極めてシンプルに実現できる．例えば（binding [name "value"] ...）と書くと，現在のスレッドについてのみ，nameが"value"にバインドされる．

3日目で学んだこと

3日目は，並行性構文について解説した．興味深い並行性構文がいくつか登場した．

参照を使うと，スレッド間の一貫性を維持しながら，可変状態を実装できる．解説では，STM（ソフトウェアトランザクショナルメモリ）を使用した．dosync関数を用いて，トランザクション内で参照に対してあらゆる変更を行った．

次に，アトムを使った．アトムは軽量の並行性構文で，データの保護という点では劣るが，使い方は簡単になる．解説では，トランザクションの外でアトムを変更した．

最後に，エージェントを用いて，処理に時間のかかる計算を実行するためのプールを実装した．エージェントは任意の関数を用いてエージェントの値を変更できるため，Ioのアクターとは異なる．エージェントは時刻スナップショットも貸す．スナップショットの値はいつでも変更できる．

セルフスタディ3日目

2日目では，高度なプログラミング抽象化を中心に解説した．3日目では，Clojureの並行性構文について説明した．以下の演習で，3日間で学んだことを試してみてほしい．

探してみよう：

- キューが空のときブロックして，キューに新しい項目が登録されるのを待つキューの実装

試してみよう：

- 参照を使ってメモリ内に口座を表すベクタを作成せよ．口座の残高を変更するdebit関数とcredit関数を作成せよ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ここでは「居眠り床屋」問題について概説する．この問題は 1965 年に Edsger Dijkstra によって作成された．内容は次のとおりだ．

- 営業中の床屋がある．
- 客は，10〜30 ミリ秒の任意の間隔でやってくる．
- 床屋の待合室には 3 つの椅子がある．
- 理容師は一人で，理容椅子は一脚用意されている．
- 理容椅子が空であれば，客が座り，理容師を起こして，髪を切ってもらう．
- 理容師が他の客の髪を切っていたら，待合室の椅子に座る．ただし待合室に空きがなければ，客は髪を切らずに黙って帰る．
- 髪を切るには 20 ミリ秒かかる．
- 髪を切ってもらった客は，立ち上がって店から出て行く．

この理髪店で，10 秒間にさばける客の数を決定するマルチスレッドプログラムを書け．

7.5　Clojure のまとめ

Clojure は，Lisp の方言としてパワーと JVM の利便性を兼ね備えている．Clojure は，JVM の既存のコミュニティ，配布プラットフォーム，コードライブラリから利点を得ている．一方，Lisp 方言であることからくる強みと制約の両方がある．

Lisp パラドックス

Clojure はおそらく，本書で紹介するなかで最も強力で柔軟な言語だ．マルチメソッドを使えば，複数のパラダイムに対応するコード（マルチパラダイムコード）を書くことができる．マクロを使えば言語を動的に再定義できる．本書に登場する言語でこのような強力な組み合わせを備えている言語は他にない．この柔軟性が信じられないくらいの強みになっている．Graham はその著作『ハッカーと画家』[Gra04] で，Lisp を使って，他のベンダーがとても及ばないような生産性を達成した新興企業の話を書いている．一部の新興コンサルタント企業も同じアプローチを採用しており，Clojure は他の言語では到底実現出来ない生産性と品質の向上を実現できると断言している．

しかし，Lisp のこうした柔軟性は，強みであると同時に弱点でもある．マクロ展開は専門家が使えば強力な機能だが，適切な考えと注意をもって臨まなければ，手痛い失敗につながる．Lisp では，数行のコードで多くの強力な抽象化を難なく適用できるが，そのために，非常にスキルの高いプログラマを除くすべての人にとって，大変習得が難しい言語になっている．

Clojure を正しく評価するには，Lisp としての側面だけでなく，Java エコシステム固有の側面やその他の独自の機能にも目を向ける必要がある．以下，Clojure の基本的な強みについて詳しく見ていこう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:17:42 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

長所

Clojure は，Java 仮想マシン上で動作する次の主要言語の座をねらっている言語の 1 つである．Clojure は実際に，次の主要言語の有力候補となっているが，それにはさまざまな理由がある．

Lisp としての質の高さ

プログラミング言語の専門家でスーパーブロガーである Tim Bray は，“Eleven Theses on Clojure.”[*10] と題する記事の中で，「Clojure は Lisp として優れている」と書いている．実際彼は Clojure のことを「これまでで最高の Lisp」とまで持ち上げている．筆者も Clojure が優れた Lisp であることには同感である．

この章では，Clojure がこのような優れた Lisp として評価される要因として Rich Hickey 氏が挙げている以下の点について説明した．

- 括弧数の削減．Clojure はシンタックスの制限を少し緩めることによって読みやすさを向上させている．具体的には，ベクタのブラケット，マップのブレース，セットの文字の組み合わせなどだ．
- エコシステム．Lisp には多くの方言があるため，どの方言をとっても利用できるサポートとライブラリ・セットが手薄になっている．この問題は，皮肉にも，もう 1 つ方言を開発することで解決できる．Clojure は，JVM を基盤としているため，より良い機能と豊富なライブラリセットを目指して開発を続ける Java プログラマの能力を活用できる．
- 制約．自制を働かせてリーダマクロを避け，Clojure のシンタックスを制限することにより，Hickey は Clojure の言語機能を事実上制限したが，有害な分離方言が発生する可能性も同時に抑えた．

Lisp をプログラミング言語として大いに賞賛している人もいるだろう．その基準からすると，Clojure は純粋に新しい Lisp と見なすことができる．その意味で，Clojure は成功しているといえる．

並行性

Clojure の並行性に対するアプローチは，並行システムの設計方法を一新する可能性を秘めている．STM は，新しい方法であるため，開発者の負担はいくらか増えるが，状態の変更が適切に保護された関数内で行われているかどうか検出されるので，開発者は心配しなくてもよいという意味で，これまでにない手法であるといえる．トランザクション内でなければ，状態を変更することはできない．

Java との統合性

Clojure は，Java との極めて高い統合性を実現している．文字列や数値といった一部の Java ネイティブの型を透過的に利用できるうえ，型ヒントによってパフォーマンスを向上させている．一方，Clojure の特長は JVM との密な統合が可能であるという点なので，

[*10] http://www.tbray.org/ongoing/When/200x/2009/12/01/Clojure-Theses

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Clojure の型は Java のアプリケーションでも自由に使える．近いうちに，Clojure のより多くの部分が JVM 内で実装されるようになるだろう．

遅延評価

Clojure には，強力な遅延評価機能が追加されている．遅延評価によって問題を簡素化できる．解説では，遅延評価によって問題の解決方法を形成する方法の一端をお見せしたにすぎない．遅延評価は，計算結果が実際に必要になるまで計算の実行を遅らせるか計算をまったく実行しないことによって計算のオーバーヘッドを大幅に削減する．最後に，問題解決を遅延させることによって，難しい問題を解決するための道具が 1 つ増える．遅延評価は，再帰，繰り返し，あるいは（遅延評価を行わない）実体を伴ったコレクションの代わりに使うことが多い．

コードとしてのデータ

プログラムはリストである．他の Lisp と同様，データはコードとして表現できる．筆者は Ruby を使ってみて，プログラムでプログラムを書くことの価値を認識することができた．このプログラムでプログラムを書くという機能は，すべてのプログラミング言語にとって最も重要な機能だと筆者は考えている．関数型プログラムでは，高階関数を用いたメタプログラミングが可能だ．Lisp では，データをコードとして評価することでこの考え方を拡張している．

短所

Clojure の目標は間違いなく汎用のプログラミング言語として使ってもらうことだ．Clojure が JVM 上で動作する言語として広く普及するかどうかはまだ分からない．Clojure は素晴らしい抽象化機能を備えているが，その多くは使いこなすのに知識が必要だ．これらの機能を本当の意味で受け入れて，効果的かつ安全に利用するには，プログラマに高い教育と能力が求められる．以下に Clojure の懸念事項をいくつか挙げる．

前置記法

コードをリスト形式で表示するのは，Lisp において最も強力な機能の 1 つだが，それには犠牲が伴う．前置記法だ[*11]．通常のオブジェクト指向言語とはシンタックスが大きく異なる．前置記法に適応するのは簡単ではない．高い記憶力が必要になるうえ，コードを，外側から内側ではなく，内側から外側に解釈する必要があるからだ．筆者は，Clojure のコードを読んでいると，あまりにも細かいレベルの話を早い段階で理解することを求められるように感じることがある．筆者の短期記憶容量では，Lisp のシンタックスを追いかけるのはどうしても厳しい．人からは，慣れれば楽になると言われているが，まだそのレベルには達していない．

[*11] Clojure には引数を左から右へくぐらせてゆくマクロ ->>と ->が用意されているため，この問題は多少は軽減されている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

読みやすさ

コードとしてのデータを実現するには，もう 1 つの犠牲を払う必要がある．それは，重苦しく感じるほどの括弧の数である．人にとっての最適化とコンピュータにとっての最適化はまったく異なる．括弧の位置と数も依然として問題である．Lisp 開発者は，各括弧が対になっているかどうかをチェックしてくれるエディタに大きく依存しているが，ツールによって読みやすさの問題が完全になくなることはない．Clojure の開発者である Rich Hickey はこの問題を改善しており，その点は称賛されるべきだが，今後もこの問題がなくなることはないだろう．

学習曲線

Clojure は豊富な機能を備えているため，覚えるには時間を要する．Lisp を動かすには高い才能と豊富な経験を持つチームが必要だ．遅延シーケンス，関数型プログラミング，マクロ展開，トランザクションメモリ，高度な問題解決のアプローチなど，すべてが習得に時間のかかる強力な概念だ．

Lisp の制限

すべての妥協には何らかの犠牲が伴う．Clojure は，JVM で動作することを選択したため，末尾再帰最適化が行えなくなった．Clojure のプログラマは不格好な recur 構文を使う必要がある．シーケンス x のサイズを計算する `(size x)` を，再帰と `loop/recur` を使って実装して，比べてみてほしい．

ユーザ定義のリーダーマクロを排除した点も重要だ．利点は明らかだ．リーダーマクロを乱用すると，言語そのものが分裂しかねない．コストも明らかだ．メタプログラミングのツールがまた 1 つ失われることになる．

とっつきにくさ

Ruby や初期の Java の最も優れた特長として，プログラミング言語としてのとっつきやすさがある．どちらも比較的簡単に始められる．一方，Clojure は開発者にかかる負担が大きい．抽象化ツールや概念があまりに多いため，開発者は圧倒されてしまう．

終わりに

Clojure の長所と短所の多くは，そのパワーと柔軟性に関するものだ．確かに Clojure は習得が難しいかもしれない．とはいえ，実は Java の開発者だって既にいろいろと苦労しているのである．例えば，Java のアプリケーションレベルの抽象化に時間を費やしたり，Spring（アスペクト指向プログラミング）を使った疎結合に注目したりしている．しかし，言語レベルでの柔軟性の向上による利点は十分に得られていない．多くの開発者は，こうした中でバランスをとってきた．筆者は，新たに並行性や複雑さが求められるなか，Java プラットフォームの存続可能性はどんどん小さくなっていくと見ている．

厳格なプログラミングモデルを求めており，言語を習得するための犠牲を払う気持ちがあるなら，Clojure は最適な言語だ．目的達成のための手段を求めており，統制され，知識と経験が豊富なチームには素晴らしい言語だと思う．Clojure を使えば，より高品質のソフトウェアをより短期間で作成できる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:19:24 under the license of University of Tokyo

第 8 章

Haskell

論理は草原でさえずる小鳥.

スポック

Haskell は，関数型プログラミングにおける多くの純粋主義者にとって，純粋と自由の象徴である．その機能は豊富で強力だが，それには代償が伴う．Haskell では，関数型プログラミングを少しかじってみるといった安易な考えは許されない．とにかく関数型プログラミングという料理を余すところなく完食することを強制される．『スタートレック』のスポックを思い浮かべてほしい．冒頭の引用[*1]は，論理と真理を崇拝するスポックの性格をよく表している．彼には，世代を超えて愛されるひたむきな純粋さがある．Scala，Erlang，および Clojure では，命令型の概念を少し取り入れることができたが，Haskell にはそのような逃げ道は用意されていない．このように純粋な関数型言語であるが故に，I/O 処理や状態を蓄積するような処理は，逆に難しくなる．

8.1 Haskell とは

いつものことだが，言語がある程度の妥協を受け入れる理由を理解するには，歴史を振り返ることから始める必要がある．1980 年代の前半から中頃にかけて，純粋関数型プログラミングの概念は複数の言語によって採用された．新しい研究を推進していた主要概念は，Clojure の章で説明した遅延評価と純粋関数型プログラミングだった．1987 年に，Functional Programming Languages and Computer Architecture 会議からグループが形成され，純粋関数型言語のオープン標準を策定することを決めた．Haskell はこのグループが 1990 年に開発し，1998 年に改訂された．現在の標準は Haskell 98 と呼ばれており，数回改訂を重ねている．これには Haskell 98 の改訂版および Haskell Prime と呼ばれる Haskell の新バージョンの定義も含まれる．このように Haskell は，最初から純粋関数型言語として，とりわけ遅延評価を重視し，最高の関数型言語の概念を組み合わせることによって開発された．

Haskell は Scala と同様に強い静的型付け言語だ．たいていの場合は型推論によって適切な型が導出されるので，すべての関数型言語のなかで最も効果的な型システムの 1 つであると広く認められている．学習を進めていくと，この型システムによって，ポリモーフィズムと非常に明解な設計が可能になっていることが分かるはずだ．

Haskell は，本書で紹介した他の概念もサポートしている．具体的には，Erlang 方式の

[*1] *Star Trek: The Original Series*（スタートレック．オリジナル・シリーズ（邦題『宇宙大作戦』)），エピソード 41/42: “I, Mudd”/“The Trouble with Tribbles” マーク・ダニエル監督制作，1967 年．カリフォルニア州バーバンク，CBS Paramount International Television，2001 年.

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

パターンマッチングとガード，Clojure 方式の遅延評価，Clojure および Erlang のリスト内包表記などだ．

Haskell は純粋関数型言語なので副作用がない．その代わり，関数は副作用を返すことができる．この副作用は後で実行できる．こうした関数の例，およびモナドと呼ばれる概念を利用して状態を保存する例については 3 日目に説明する．

最初の 2 日間で，式，関数の定義，高階関数など，典型的な関数型プログラミングの概念について説明する．また Haskell の型モデルも紹介するが，そこでいくつか新しい概念が登場することになる．3 日目は範囲を広げて，パラメータ化された型システムとモナドについて見ていく（これらの概念は理解が難しいかもしれない）．では始めよう．

8.2　1 日目：論理的

スポックと同様，Haskell の中心的な概念は理解しやすいことが分かる．関数の定義は厳密に行う．同じ入力引数を与えれば，何度でも同じ出力引数が返される．本書では，GHC（Glasgow Haskell Compiler）6.12.1 を使う．このバージョンは多くのプラットフォームで利用可能だが，他の実装もある．いつものとおり，コンソールを起動してみる．ghci と入力してほしい．

```
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
```

パッケージがいくつかロードされる．ロードが完了すると，コマンドを入力できる状態になる．

式とプリミティブ型

Haskell の型システムについてはもう少し後で説明する．ここではプリミティブ型に絞って説明する．他の多くの言語と同様，数値と簡単な式から始め，その後，より高度な型（関数など）の説明に移ることにする．

数値

では，いつもの手順で始めよう．次の式を入力してみてほしい．

```
Prelude> 4
4
Prelude> 4 + 1
5
Prelude> 4 + 1.0
5.0
Prelude> 4 + 2.0 * 5
14.0
```

演算の優先順は，期待どおりに動作する．

```
Prelude> 4 * 5 + 1
21
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Prelude> 4 * (5 + 1)
24
```

演算を括弧で囲んでグループ化することもできる．以上，いくつかの型の数値を試してみた．次に，文字データについて見てみよう．

文字データ

文字列はダブルクオートで囲んで指定する．次に例を示す．

```
Prelude> "hello"
"hello"
Prelude> "hello" + " world"

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of `+' at <interactive>:1:0-17
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: "hello" + " world"
    In the definition of `it': it = "hello" + " world"
Prelude> "hello" ++ " world"
"hello world"
```

連結演算子は，+ ではなく ++ である点に注意してほしい．1 つの文字は次のように指定する．

```
Prelude> 'a'
'a'
Prelude> ['a', 'b']
"ab"
```

文字列は単なる文字のリストである点に注意してほしい．次に，ブール値について簡単に見ておこう．

ブール値

ブール型もプリミティブ型で，本書で取り上げた他のほとんどの中置表記法言語のブール型と同じように動作する．

```
Prelude> (4 + 5) == 9
True
Prelude> (5 + 5) /= 10
False
```

if/then 文を試してみる．

```
Prelude> if (5 == 5) then "true"

<interactive>:1:23: parse error (possibly incorrect indentation)
```

本書で取り上げた他の言語と大きく異なる部分が初めて出てきた．Haskell ではインデントが重要な意味を持つ．このエラーでは，後続の行が正しくインデントされていないと言っている．インデント構造については後で見る．Haskell にはインデントのパターンを制御するレイアウト規則が用意されているが，これについては触れない．ここに掲載したコードをまねた述語インデント方式に従っておけば問題はない．完全な if/then/else

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

文を実行してみよう．

```
Prelude> if (5 == 5) then "true" else "false"
"true"
```

Haskell では，if は制御構文ではなく関数である．したがって，他の関数と同じように値を返す．真偽値をいくつか試してみよう．

```
Prelude> if 1 then "true" else "false"

<interactive>:1:3:
    No instance for (Num Bool)
      arising from the literal ‘1’ at <interactive>:1:3
    ...
```

Haskell は強く型付けされた言語である．if は厳密にブール型の値をとる．もう 1 つ，型衝突を起こさせてみよう．

```
Prelude> "one" + 1

<interactive>:1:0:
    No instance for (Num [Char])
      arising from a use of ‘+’ at <interactive>:1:0-8
      ...
```

このエラーメッセージは Haskell の型システムの片鱗を覗かせる最初の例だ．具体的には，「Num 引数の後に文字のリスト [Char] がくる関数 + は存在しない」と言っている．ここで型の指定は一切していない点に注意してほしい．Haskell は，さまざまなヒントに基づいて型を推論する．Haskell がどのようにして型を推論しているかは，いつでも確認できる．:t と指定するか，同じ効果のある t オプションをオンにすればよい．次に例を示す．

```
Prelude> :set +t
Prelude> 5
5
it :: Integer
Prelude> 5.0
5.0
it :: Double
Prelude> "hello"
"hello"
it :: [Char]
Prelude> (5 == (2 + 3))
True
it :: Bool
```

これで，式を評価するたびに，その式が返す型を確認できる．ただし，:t を数値と一緒に使うと混乱するので注意が必要だ．これは，数値とコンソール間の相互作用と関連している．次のように，:t 関数を使うようにすること．

```
Prelude> :t 5
5 :: (Num t) => t
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

先ほどは it :: Integer だったのに，今度は型が異なっている．コンソールは，:set +t を指定しない限り，数値をできる限り総称的に処理しようとする．つまり，純粋な型ではなく，クラスが返される．クラスとは，同種の型の集まりを記述したものである．クラスについては，8.4「クラス」(p. 259)で詳しく説明する．

関数

Haskell プログラミングパラダイム全体の中心となるのは関数である．Haskell は強く静的な型付けを行うため，各関数は 2 つの部分，すなわち型指定（省略可能）と実装に分けて指定する．これから他の言語で既に見た概念について駆け足で説明する．しっかりつかまってついてきてほしい．

基本的な関数の定義

Haskell の関数は，従来より，型宣言と関数宣言の 2 つの部分からなる．

最初はコンソール内で関数を定義していく．let 文を使って値を実装にバインドする．関数を定義する前に let を試してみよう．Lisp と同様，Haskell でも let はローカルスコープ内で変数を関数にバインドする．

```
Prelude> let x = 10
Prelude> x
10
```

Haskell のモジュールをコーディングするときは，次のように関数を宣言する．

```
double x = x * 2
```

コンソールでは，let を使ってローカルスコープ内に関数を割り当てれば，使えるようになる．引数を倍にして返す簡単な関数 double の例を示す．

```
Prelude> let double x = x * 2
Prelude> double 2
4
```

ここから，ファイルにプログラムを保存する方法に切り替える．これにより，複数行の定義を書けるようになる．GHC では，完全な double の定義は次のようになる．

▶ code/haskell/double.hs

```
module Main where

    double x = x + x
```

Main という名前のモジュールを追加している点に注意してほしい．Haskell では，モジュールとは関連するコードを同じスコープ内に集めたものである．Main モジュールは特別で，トップレベルのモジュールだ．ここでは，double 関数を例に説明する．Main をコンソールにロードして次のように使う．

```
Prelude> :load double.hs
[1 of 1] Compiling Main             ( double.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
10
```

ここまで，型を指定していない．これが許されているのは，Haskell が我々に代わって型を推論してくれているからだ．もちろん，各関数には型記述が存在している．型記述を明示的に指定した関数の例を示す．

▶ code/haskell/double_with_type.hs

```
module Main where

    double :: Integer -> Integer
    double x = x + x
```

前と同じようにロードして使うことができる．

```
[1 of 1] Compiling Main             ( double_with_type.hs, interpreted )
Ok, modules loaded: Main.
*Main> double 5
10
```

新しい関数に関連付けられている型は次のようにして確認できる．

```
*Main> :t double
double :: Integer -> Integer
```

この定義は，関数 double は Integer 型の引数（最初の Integer）をとり，Integer 値を返すことを意味する．

この型記述は制限的である．先ほど定義した型のない double の実行結果を見ると，まったく異なる結果になっている．

```
*Main> :t double
double :: (Num a) => a -> a
```

見てのとおり，推論された型がまったく異なる．この例では，a は型変数である．この定義は，「関数 double は，ある型 a の引数を 1 つとり，同じ型 a の値を返す」となる．前の制限的な定義を改善したこの定義により，この関数は，+ 関数をサポートする任意の型に適用できるようになる．それでは，そろそろレベルを上げて，もう少し面白いものを実装してみよう．まずは，階乗計算からだ．

再帰

再帰を利用した簡単な例から始めよう．コンソール内で階乗計算を行う再帰的なワンライナー（1 行プログラム）を以下に示す．

```
Prelude> let fact x = if x == 0 then 1 else fact (x - 1) * x
Prelude> fact 3
6
```

これは初版だ．文章にすると，「x の階乗は，x が 0 のとき 1，それ以外のときは fact (x - 1) * x である」となる．これから，このコードを改良していく．パターンマッチングを導入するともう少し良くなる．実は，このシンタックスは，見かけも動作も Erlang のパターンマッチングと非常によく似ている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:19:24 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/factorial.hs

```
module Main where
    factorial :: Integer -> Integer
    factorial 0 = 1
    factorial x = x * factorial (x - 1)
```

定義は3行だ．1行目で，引数と，返す結果の型を宣言している．2行目と3行目は，入力された引数とのパターンマッチングの結果に応じて呼び出される異なる関数定義だ．0のfactorialは1であり，nのfactorialはx = x * factorial (x - 1)だ．この定義は，数学の階乗定義とほぼ一致している．この例では，パターンの指定順序が重要になる．最初にマッチした関数定義が呼ばれるからだ．順序を変更する場合は，ガードを使用する必要がある．Haskellでは，ガードは引数の値を制限する条件として使われる．次に例を示す．

▶ code/haskell/fact_with_guard.hs

```
module Main where
    factorial :: Integer -> Integer
    factorial x
        | x > 1 = x * factorial (x - 1)
        | otherwise = 1
```

この例のガードでは，左側にブール値を，右側に適用される関数を指定している．ガードが満たされると，対応する関数が呼び出される．ガードはよくパターンマッチングの代わりに使われる．ここでは，再帰の基本条件（終了条件）を起動するためにガードを用いている．

タプルとリスト

他の言語でも見たように，Haskellは末尾再帰最適化によって再帰を効率的に処理している．Haskellでフィボナッチ数列を生成するコードをいくつか異なる方法で書いてみよう．最初は最も簡単なバージョンだ．

▶ code/haskell/fib.hs

```
module Main where
    fib :: Integer -> Integer
    fib 0 = 1
    fib 1 = 1
    fib x = fib (x - 1) + fib (x - 2)
```

見てのとおり極めて単純だ．fib 0またはfib 1は1，fib xはfib (x - 1) + fib (x - 2)だ．しかしこの解法は非効率的だ．もう少し効率的な解法を考えてみよう．

タプルを使ったプログラミング

処理効率を上げるにはタプルを用いる．タプルとは固定数項目のコレクションだ．Haskellでは，カンマで区切られた要素を括弧で囲むことによってタプルを表す．次の実装では，連続するフィボナッチ数からなるタプルを作成している．その際，再帰の終了条件にカウンタを使っている．以下に基本的な解法を示す．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
fibTuple (x, y, 0) = (x, y, 0)
fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)
```

fibTuple は 3 要素のタプルを引数として受け取り，3 要素のタプルを返す．ここで，単一の引数として 3 要素のタプルをとるのと，3 つの引数をとるのとでは意味が異なるので注意してほしい．この関数を使うには，2 つの数値 0 と 1 から再帰を始める．カウンタをカウントダウンするたびに，最初の 2 つの要素に，フィボナッチ数列内の連続する数値が入っていく．fibTuple (0, 1, 4) を呼び出したときの様子は次のようになる．

- fibTuple (0, 1, 4)
- fibTuple (1, 1, 3)
- fibTuple (1, 2, 2)
- fibTuple (2, 3, 1)
- fibTuple (3, 5, 0)

このプログラムは次のように実行する．

```
Prelude> :load fib_tuple.hs
[1 of 1] Compiling Main             ( fib_tuple.hs, interpreted )
Ok, modules loaded: Main.
*Main> fibTuple(0, 1, 4)
(3, 5, 0)
```

1 番目の要素が答えとなる．答えだけ取り出すには次のようにする．

```
fibResult :: (Integer, Integer, Integer) -> Integer
fibResult (x, y, z) = x
```

パターンマッチングを用いて 1 番目の要素を取り出した．これにより，次のように呼び出し形式を単純化できる．

```
fib :: Integer -> Integer
fib x = fibResult (fibTuple (0, 1, x))
```

この関数は，2 つのヘルパー関数を使って，非常に高速なフィボナッチ数列ジェネレータを構築している．以下にプログラム全体を示す．

▶ code/haskell/fib_tuple.hs

```
module Main where
    fibTuple :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
    fibTuple (x, y, 0) = (x, y, 0)
    fibTuple (x, y, index) = fibTuple (y, x + y, index - 1)

    fibResult :: (Integer, Integer, Integer) -> Integer
    fibResult (x, y, z) = x

    fib :: Integer -> Integer
    fib x = fibResult (fibTuple (0, 1, x))
```

実行結果（一瞬で表示される）は次のようになる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
*Main> fib 100
354224848179261915075
*Main> fib 1000
    43466557686937456435688527675040625802564660517371780
    40248172908953655541794905189040387984007925516929592
    25930803226347752096896232398733224711616429964409065
    33187938298969649928516003704476137795166849228875
```

次に，別のアプローチを用いて関数を合成してみよう．

タプルと合成

ある関数の結果を別の関数に渡して関数をつなげることによって関数を組み合わせる必要がある場合がある．以下に，リストの tail の head をマッチングすることによってリストの 2 番目の要素を取り出す例を示す．

```
*Main> let second = head . tail
*Main> second [1, 2]
2
*Main> second [3, 4, 5]
4
```

ここではコンソール内で関数を定義している．second = head . tail は，second lst = head (tail lst) と等価である．ある関数の結果を別の関数に渡すのと同じということだ．この機能を用いて，別の方法でフィボナッチ数列を生成してみよう．前と同様，1 組のペアを計算する．ただし，今度はカウンタはなしだ．

```
fibNextPair :: (Integer, Integer) -> (Integer, Integer)
fibNextPair (x, y) = (y, x + y)
```

数列内の 2 つの数を与えることにより，常に，その次の数を計算することができる．次の仕事は，数列内の次の項を再帰的に計算することだ．

```
fibNthPair :: Integer -> (Integer, Integer)
fibNthPair 1 = (1, 1)
fibNthPair n = fibNextPair (fibNthPair (n - 1))
```

基本条件は，n が 1 のときの (1, 1) だ．あとは簡単で，直前の項に基づいて数列の次の項を計算するだけだ．こうして数列内の任意の 1 組を求めることができる．

```
*Main> fibNthPair(8)
(21,34)
*Main> fibNthPair(9)
(34,55)
*Main> fibNthPair(10)
(55,89)
```

あとは各ペアの最初の項を取り出せばいい．関数合成を便利に使って，最初の要素を取り出す fst と，1 組のペアを作り出す fibNthPair を組み合わせよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/fib_pair.hs

```
module Main where
    fibNextPair :: (Integer, Integer) -> (Integer, Integer)
    fibNextPair (x, y) = (y, x + y)

    fibNthPair :: Integer -> (Integer, Integer)
    fibNthPair 1 = (1, 1)
    fibNthPair n = fibNextPair (fibNthPair (n - 1))

    fib :: Integer -> Integer
    fib = fst . fibNthPair
```

別の言い方をすると，n 番目のタプルの最初の要素を取り出したということだ．以上，タプルについて簡単に見てきた．次に，リストを用いていくつか問題を解いてみよう．

リストの探索

リストについては既に多くの異なる言語で見てきたので，一から説明することはしない．その代わり，基本的な再帰の例についてざっと説明したあと，これまでに登場していない新しい関数をいくつか紹介しよう．リストをヘッドとテールに分解する方法は，let 文やパターンマッチングなど，任意のバインディングで使える．

```
let (h:t) = [1, 2, 3, 4]
*Main> h
1
*Main> t
[2,3,4]
```

この例では，リスト [1, 2, 3, 4] を (h:t) にバインドしている．この構文は，Prolog，Erlang，Scala に出てきたさまざまな head|tail 構文と同じだと考えてよい．この構文を使うことで，いくつかの簡単な再帰定義を行うことができる．以下に，リストの size（サイズ）と prod（直積）を求める関数を示す．

▶ code/haskell/lists.hs

```
module Main where
    size [] = 0
    size (h:t) = 1 + size t

    prod [] = 1
    prod (h:t) = h * prod t
```

ここでは Haskell の型推論を用いてこれらの関数の型を決めているが，関数の内容は明らかだ．リストのサイズは，1 にテールのサイズを加えたものだ．

```
Prelude> :load lists.hs
[1 of 1] Compiling Main
    ( lists.hs, interpreted )
Ok, modules loaded: Main.
*Main> size "Fascinating."
12
```

zip はリストを組み合わせる強力な方法だ．以下に使い方を示す．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
*Main> zip "kirk" "spock"
[('k','s'),('i','p'),('r','o'),('k','c')]
```

2要素のタプルが生成された．次のように，リストをzipすることもできる．

```
Prelude> zip ["kirk", "spock"]
[("kirk","spock")]
```

このようにすれば，2つのリストを効率的に組み合わせることができる．

ここまで見てきたHaskellの機能は，他の言語で説明した機能と驚くほどよく似ている．ここからは少し高度な構文について見ていくことにする．具体的には，範囲やリスト内包表記などの高度なリスト構文について説明する．

リストを作り出す

ここまで，リストを再帰的に処理する方法についていくつか見てきた．このセクションでは，新しいリストを作り出す方法をいくつか紹介する．具体的には，再帰，範囲，リスト内包表記の3つについて説明する．

再帰

リストを構成する最も基本的な部品は:演算子である．この演算子は，ヘッドとテールを結合してリストを作る．再帰関数を呼び出すとき，パターンマッチングで，この演算子を逆方向に使用する例については既に見た．次の例では，letの左辺に:を使っている．

```
Prelude> let h:t = [1, 2, 3]
Prelude> h
1
Prelude> t
[2,3]
```

:演算子は，リストを分解する代わりに，リストを構築する場合にも使える．
次のような具合だ．

```
Prelude> 1:[2, 3]
[1,2,3]
```

リストには異なる型を混在させることはできない．例えば，リストを整数のリストに追加することはできない．

```
Prelude> [1]:[2, 3]

<interactive>:1:8:
    No instance for (Num [t])
      arising from the literal `3' at <interactive>:1:8
```

しかし，リストをリストのリスト，あるいは空リストに追加することはできる．

```
Prelude> [1]:[[2], [3, 4]]
[[1],[2],[3,4]]
Prelude> [1]:[]
[[1]]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

以下に，リストを構築する例を示す．リストから偶数を取り出して返す関数を作成したいとする．この関数は，リストの構築を使って書くことができる．

▶ code/haskell/all_even.hs

```
module Main where
    allEven :: [Integer] -> [Integer]
    allEven [] = []
    allEven (h:t) = if even h then h:allEven t else allEven t
```

この関数は整数のリストを引数にとり，偶数のリストを返す．allEven に空リストを渡すと，空リストが返される．ヘッドが偶数のリストの場合は，そのヘッドと，テールに allEven を適用した結果を連結する．ヘッドが奇数の場合は，それを破棄し，テールに allEven を適用した結果のみを返す．簡単だ．リストを構築する他の方法をいくつか見てみよう．

範囲と合成

Ruby や Scala と同様，Haskell にもファーストクラスの範囲とそれらをサポートするいくつかのシンタックスシュガーが用意されている．Haskell の範囲は，次のように範囲の両端を指定する簡単な形式で表現する．

```
Prelude> [1..2]
[1,2]
Prelude> [1..4]
[1,2,3,4]
```

両端を指定すると，範囲が計算される．デフォルトの増分は 1 だ．デフォルトの増分では終点に到達できない場合はどうなるのだろうか？

```
Prelude> [10..4]
[]
```

上のように，空リストが返される．増分を指定するには，リスト内の 2 番目の要素も指定する．

```
Prelude> [10, 8 .. 4]
[10,8,6,4]
```

リストの要素は小数になっても構わない．

```
Prelude> [10, 9.5 .. 4]
[10.0,9.5,9.0,8.5,8.0,7.5,7.0,6.5,6.0,5.5,5.0,4.5,4.0]
```

範囲は，数列を作成するためのシンタックスシュガーだ．数列には終点はなくてもよい．Clojure と同様，数列の一部の項を抜き出すことができる．

```
Prelude> take 5 [ 1 ..]
[1,2,3,4,5]
Prelude> take 5 [0, 2 ..]
[0,2,4,6,8]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

遅延数列については2日目で詳しく説明する．ここでは，リストを自動的に作成する別の方法，すなわちリスト内包表記について見ていこう．

リスト内包表記

リスト内包表記についてはErlangの章で最初に説明した．リスト内包表記はHaskellでも同じように動作する．左側には式を書く．右側には，Erlangのときと同じように，ジェネレータやフィルタを書く．いくつか例を見てみよう．リスト内のすべての項を2倍するには，次のように書く．

```
Prelude> [x * 2 | x <- [1, 2, 3]]
[2,4,6]
```

文章にすると，「[1, 2, 3]から取り出した各要素xに対してx * 2を計算した結果をリストとして返せ」となる．

Erlangと同様，リスト内包表記内でパターンマッチングを使うこともできる．例えば，多角形を表す点のリストがあるとき，その多角形を対角線を対称軸として対称移動したいとする．それには次のようにxとy入れ換えればよい．

```
Prelude> [ (y, x) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(2,1),(3,2),(1,3)]
```

多角形を水平方向に反転するには，この多角形の場合4からxを引けばよい．

```
Prelude> [ (4 - x, y) | (x, y) <- [(1, 2), (2, 3), (3, 1)]]
[(3,2),(2,3),(1,1)]
```

組み合わせを計算することもできる．カーク，スポック，マッコイの3人のクルーから2人を上陸部隊として選ぶとき，可能な組み合わせをすべて見つけるために，まず次のようにしてみる．

```
Prelude> let crew = ["Kirk", "Spock", "McCoy"]
Prelude> [(a, b) | a <- crew, b <- crew]
[("Kirk","Kirk"),("Kirk","Spock"),("Kirk","McCoy"),
("Spock","Kirk"),("Spock","Spock"),("Spock","McCoy"),
("McCoy","Kirk"),("McCoy","Spock"),("McCoy","McCoy")]
```

まずまずだが，重複要素が排除されていない．そこで次のようにリスト内包表記をフィルタリングする条件を追加する．

```
Prelude> [(a, b) | a <- crew, b <- crew, a /= b]
[("Kirk","Spock"),("Kirk","McCoy"),("Spock","Kirk"),
("Spock","McCoy"),("McCoy","Kirk"),("McCoy","Spock")]
```

だいぶ良くなったが，順序は関係ない．そこで，ソートされた組だけを残し，残りを無視するようにする．

```
Prelude> [(a, b) | a <- crew, b <- crew, a < b]
[("Kirk","Spock"),("Kirk","McCoy"),("McCoy","Spock")]
```

このように短く簡単なリスト内包表記で，求める答えが得られた．リスト内包表記は，リストを迅速に作成および変換するための優れた道具である．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Philip Wadler 氏へのインタビュー

Haskell の主な機能について説明したところで，Haskell の設計委員会のメンバーの話を聞いてみることにしよう．エジンバラ大学理論コンピュータ科学学科教授の Philip Wadler は，Haskell だけでなく，Java，XQuery のコントリビュータとしても積極的に活動している．彼は，アバヤ・ラボ，ベル研，グラスゴー大学，チャルマーズ工科大学，オックスフォード大学，ゼロックス・パロアルト研究所，スタンフォード大学での勤務または研究実績がある．

Bruce Tate　あなたのチームで Haskell を開発した理由を聞かせてください．

Philip Wadler　1980 年代後半には，関数型言語の設計と実装を行うグループが多数存在していました．そこで，そうしたグループがばらばらに活動するより，協力したほうが力を発揮できることに気づいたのです．最初に設定した目標は簡単なものではありませんでした．我々は，研究の基盤となり，教えやすく，一般企業での使用にも耐える言語を作りたかったのです．その辺りの経緯については，我々のグループが History of Programming Languages 会議に寄稿した論文に詳しく書かれています．この論文は Web 上で公開されています[*2]．

Bruce Tate　Haskell の機能で最も気に入っているのはどの部分ですか？

Philip Wadler　私は，リスト内包表記でプログラミングするのが楽しくてしかたありません．リスト内包表記が，ようやく，Python など，他の言語にも広まってきたのはうれしいことです．
クラスはジェネリックなプログラミングをシンプルな形で実現します．データ型を定義したあと，単に `derived` というキーワードを追加するだけで，値を比較したり，文字列との間で値を変換したりするルーチンを書くことができます．この機能はとても便利なので，他の言語を使っているときにも同じ機能が欲しくなってしまいます．
良いプログラミング言語というのは，目の前にある仕事を片付けるための他のプログラミング言語を埋め込むことによって拡張できます．Haskell は，他の言語を埋め込むには素晴らしいツールです．Haskell では，遅延評価，ラムダ式，モナド，アロー記法，クラス，表現力に富む型システム，テンプレートなど，さまざまな方法で言語を拡張できます．

Bruce Tate　一から作り直せるとしたら，どの部分を変更しますか？

Philip Wadler　昨今では，分散処理がますます重要になってきているため，互いに値を受け渡しながら複数のマシン上で動作する複数のプログラムの実現に注力する必要があります．値を送信するときは，おそらく，評価することによって値を生成するプログラム（およびプログラムのすべての自由変数の値）ではなく値そのもの（先行評価）が欲しいでしょう．ですから，分散プログラミングの世界では，デフォルトでは先行評価にしておいて，必要なときに簡単に遅延評価に変更できるようにしておくほうがよいと考えています．

[*2] http://www.haskell.org/haskellwiki/History_of_Haskell

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Bruce Tate Haskellを用いたソリューションで，最も面白いと思ったものを教えてください．

Philip Wadler ユーザの皆さんが発見するHaskellの使い方にはいつも驚かされます．数年前，自然言語処理にHaskellが使われているのを見て驚いたのを覚えています．その数年後には，AIDS治療の研究でタンパク質の畳み込み解析に使われているのを見ました．Haskellコミュニティのページを覗いてみると，そこには40もの製造業向けアプリケーションの一覧が掲載されています．今では，ABN Amro，Credit Suisse，Deutsche Bank，Standard Charteredなど，金融分野にも多数がユーザがいます．Facebookは，PHPで書かれたコードを更新するための社内ツールとしてHaskellを使っています．私が最も気に入っているのは，Haskellをガベージコレクションに使った例です．と言っても，ソフトウェアの世界の話ではなく，本当のゴミ収集のことです．つまり，ゴミ収集トラック用のプログラミングエンジンにHaskellが使われていたのです．

1日目で学んだこと

Haskellは関数型プログラミング言語だ．他の言語とは異なるHaskell最大の特徴は，純粋な関数型言語であるという点だ．ある関数に同じ引数を指定すると，常に，同じ結果が返される．副作用はない．1日目の今日は，本書で紹介した他の言語にも用意されていた機能について説明した．

まず，基本的な式と簡単なデータ型について説明した．値が変わる変数への代入というものが存在しないので，再帰を使って，いくつかの簡単な数学の関数を定義し，リストを処理した．基本的なHaskellの式を覚え，それらを関数に取り込んだ．ErlangやScalaと同様のパターンマッチングとガードについても説明した．また，Erlangのときと同じように，リストとタプルを基本的なコレクションとして使った．

最後に，リストの構築について説明し，その際，リスト内包表記，範囲，さらには遅延数列などについても触れた．これらのいくつかの概念を，次の演習で実際に使ってみよう．

セルフスタディ1日目

ここまで読み進めてきた皆さんは，他のすべての関数型言語もひととおり学習されたと思うので，関数型プログラムを書くのも楽になっているはずだ．このセクションでは少し高度な問題を解いてもらおう．

探してみよう：

- Haskellのwiki
- お使いのコンパイラをサポートするHaskellのオンライングループ

試してみよう：

- `allEven`にはいろいろな書き方がある．何通りの書き方を見つけられるか試してみよ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- リストを受け取って，同じリスト（の各項の並び）を逆順にして返す関数を書け．
- black，white，blue，yellow，red から 2 色を取った組み合わせを表す 2 要素のタプルをすべて作成する関数を書け．順番は関係ないので，例えば，(black，blue) と (blue，black) はどちらか一方のみを含めるようにすること．
- リスト内包表記を用いて，子供が使う乗算表を作れ．この表は，3 要素のタプルのリストとして表現できる．その場合，各タプルの最初の 2 要素は 1〜12 の整数，3 番目の要素は最初の 2 要素を掛け合わせた値となる．
- Haskell を使って，地図塗り分け問題（4.2「地図の塗り分け」(p. 78)）を解いてみよ．

8.3　2 日目：スポックの大きな強み

人によっては，その性格の最も良いところが，かなり長い間，他の人から気づかれないことがあるが，スポックに関して言えば，彼の強みを理解するのは容易だ．彼は，優秀で，常に論理的で，いつも期待どおりに動いてくれる．Haskell の強みもやはり，その予測可能性と論理のシンプルさにある．多くの大学がプログラムでの推論に関する講義で Haskell を教えている．Haskell は，命令型言語に比べて，正しさの証明をはるかに簡単に行うことができる．このセクションでは，予測可能性をさらに高める実用的な概念について掘り下げていく．まずは，高階関数から始めて，次に Haskell における高階関数の組み合わせ方法を説明する．そこから，部分適用関数とカリー化へと話がつながる．最後に遅延評価について説明する．たっぷり 1 日はかかりそうだ．さっそく始めよう．

高階関数

本書で取り上げたすべての言語には，高階プログラミングという概念を取り入れているが，とりわけ Haskell は，この高階関数という概念に大きく依存している．ここでは，無名関数についてざっと説明したあと，それをリスト操作用の多数の組み込み関数に適用してみる．既に紹介済みの概念でもあるし，説明しなければならない範囲も広いので，他の章の説明よりぐんとスピードアップさせてもらう．まずは，無名関数からだ．

無名関数

期待どおり，Haskell の無名関数のシンタックスは拍子抜けするくらい簡単だ．(\param1 .. paramn -> function_body) と書くだけだ．いくつか試してみよう．

```
Prelude> (\x -> x) "Logical."
"Logical."
Prelude> (\x -> x ++ " captain.") "Logical,"
"Logical, captain."
```

無名関数は，単独ではたいして役に立たないが，他の関数と組み合わせると，非常に強力な道具となる．

map と where

最初の例は，第 1 引数を返すだけの無名関数だ．次の例では，文字列を末尾に付けている．他の言語でも見たように，無名関数はリストライブラリには重要な機能だ．Haskell

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:21:05 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

には次のようにmap関数が用意されている．

```
map (\x -> x * x) [1, 2, 3]
```

この例では，map関数を無名関数とリストに適用している．mapはリストの各項に無名関数を適用し，結果をコレクションとして返す．驚くようなことは何もない．しかし，この形式は，一度に理解するには少し難しいかもしれない．そこで，これと同じ処理を1つの関数にまとめてから，無名関数をローカルスコープの関数に変えてみよう．次のようになる．

▶ code/haskell/map.hs

```
module Main where
    squareAll list = map square list
        where square x = x * x
```

まずlistという引数をとる関数squareAllを宣言する．次にmapを使って，squareという関数をlist内のすべての項に適用する．そして，新しい機能whereを用いて，squareのローカル版を宣言する．関数をwhereにバインドする必要はない．また，任意の変数をバインドできる．whereを用いた例は，これからいたるところに登場する．では上の関数を実行してみよう．

```
*Main> :load map.hs
[1 of 1] Compiling Main             ( map.hs, interpreted )
Ok, modules loaded: Main.
*Main> squareAll [1, 2, 3]
[1,4,9]
```

セクションと呼ばれる部分的な関数の一種と共にmapを使うこともできる．

```
Prelude> map (+ 1) [1, 2, 3]
[2,3,4]
```

(+ 1)は，実は，部分適用関数だ．+関数は2つの引数をとるが，ここでは1つしか与えていない．その結果，(x + 1)のように単一の引数xをとる関数が得られる．

filter，foldl，foldr

次によく使う関数としてfilterがある．filterは，リスト内の要素にテストを適用する．次に例を示す．

```
Prelude> odd 5
True
Prelude> filter odd [1, 2, 3, 4, 5]
[1,3,5]
```

Clojureや Scalaと同じように，左または右に畳み込むこともできる．Haskellでもfoldlとfoldrをさまざまな形で使う．

```
Prelude> foldl (\x carryOver -> carryOver + x) 0 [1 .. 10]
55
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

この例では，キャリーオーバーの初期値を 0 として，リストの各要素に関数を適用する．その際，関数が返す結果を carryOver 引数として，リストの各要素をもう 1 つの引数として与える．畳み込みにはもう 1 つの形式がある．この形式は，演算子を使って畳み込む場合に便利だ．

```
Prelude> foldl1 (+) [1 .. 3]
6
```

この例では，+ 演算子を，2 つの引数をとり整数値を返す純粋な関数として用いている．結果は，次の式を評価した場合と同じになる．

```
Prelude> 1 + 2 + 3
6
```

foldr1 を使えば，右から左に畳み込むこともできる．

想像はつくと思うが，Haskell では，リスト操作用関数のライブラリに，上記以外にも多数の関数が用意されている．それらの多くは高階関数を使ったものだ．高階関数の説明に章全体を割くことはしないので，あとは自分でいろいろと調べてほしい．次に，Haskell で関数を組み合わせる方法の説明に移ろう．

部分適用関数とカリー化

前のセクションでは関数の合成と部分適用関数について簡単に説明した．これらの概念は Haskell にとって重要かつ中心となる部分なので，ここでもう少し時間を割いて説明しておく．

Haskell のすべての関数は 1 つの引数をとる．というと，次のような疑問を持つかもしれない．「それが本当なら，+ のように 2 つの数値を加算する関数はどのようにして書くのか」．

実際，Haskell のすべての関数は 1 つの引数をとるのである．型のシンタックスを単純化するため，prod という関数を作成してみよう．

```
Prelude> let prod x y = x * y
Prelude> prod 3 4
12
```

関数を作成して，それが動作するのを確認した．この関数の型を調べてみよう．

```
Prelude> :t prod
prod :: (Num a) => a -> a -> a
```

Num a => の部分は，「以下の型記述で，a は Num（クラス）の型である」という意味だ．残りの部分は前にも出てきたが，そのときは，説明を簡単にするために正確な意味については触れなかった．この辺できちんと説明しておこう．Haskell では，ある概念を用いて，複数の引数をとる 1 つの関数を，1 つの引数をとる複数の関数に分割する．これを実現するために使われるのが，部分適用という手法だ．

用語に混乱しないようにしてほしい．部分適用とは，全部ではなく一部の引数をバインドするという意味だ．例えば，prod を部分適用して，別の関数を作成できる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
Prelude> let double = prod 2
Prelude> let triple = prod 3
```

まず，これらの関数の左辺を見てほしい．2引数の関数 prod を定義しているが，最初の引数だけを与えている．prod 2 の計算は簡単だ．元の関数 prod x y = x * y の x を2で置換すると，double y = 2 * y となる．これらの関数は期待どおりに動く．

```
Prelude> double 3
6
Prelude> triple 4
12
```

これで疑問が解けた．Haskell は，prod 2 4 を計算するとき，実は (prod 2) 4 を計算しているのである．手順は次のとおりだ．

- 最初に，prod 2 を実行して，関数 (\y -> 2 * y) を得る．
- 次に，(\y -> 2 * y) 4，すなわち 2 * 4 を実行して，8 を得る．

このプロセスはカリー化と呼ばれる．Haskell で複数の引数をとる関数は，カリー化されたものとして定義される．これにより，柔軟性が向上し，シンタックスは単純化される．実際には，ほとんどの場合，カリー化されていることを意識する必要はない．というのは，カリー化されていてもいなくても，関数の値は同じだからだ．

遅延評価

Clojure のシーケンスライブラリと同じように，Haskell も遅延評価を多用している．遅延評価を用いると，無限リストを返す関数を作成できる．リスト構成子を使って無限リストを作ることはよくある．無限範囲を作る例を見てみよう．開始値は x，増分は y だ．

▶ code/haskell/my_range.hs

```
module Main where
    myRange start step = start:(myRange (start + step) step)
```

シンタックスは奇妙だが，全体の結果は美しい．範囲の開始値と増分を引数にとる myRange という関数を作成している．リストの合成を使って，ヘッドが start で，テールが (myRange (start + step) step) のリストを作っている．myRange 1 1 を次々に評価していくと，次のような結果が得られる．

- 1:myRange (2 1)
- 1:2:myRange (3 1)
- 1:2:3:myRange (4 1)

以降同様に続く．

この再帰は無限に続くので，通常は，再帰を制限する他の関数と組み合わせて使う．最初に my_range.hs をロードするのを忘れないように．

```
*Main> take 10 (myRange 10 1)
[10,11,12,13,14,15,16,17,18,19]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
*Main> take 5 (myRange 0 5)
[0,5,10,15,20]
```

一部の再帰関数はリスト合成構文を使うことによって，より効率的に動作するようになる．以下に，遅延評価とリスト合成を用いてフィボナッチ数列を作成する例を示す．

▶ code/haskell/lazy_fib.hs

```
module Main where
    lazyFib x y = x:(lazyFib y (x + y))

    fib = lazyFib 1 1

    fibNth x = head (drop (x - 1) (take (x) fib))
```

最初の関数は，すべての項が直前の 2 つの項の和となるような数列を作っている．これで数列はできたのだが，API を改良したい．正式なフィボナッチ数列にするには数列を 1 と 1 で始める必要がある．そこで，fib は lazyFib に最初の 2 項を種として与える．最後に，もう 1 つヘルパー関数を用意する．この関数を使うと，ユーザは drop と take を用いて数列の中の 1 つの項を取り出すことができる．では上記の関数を実際に使ってみよう．

```
*Main> take 5 (lazyFib 0 1)
[1,1,2,3,5]
*Main> take 5 (fib)
[1,1,2,3,5]
*Main> take 5 (drop 20 (lazyFib 0 1))
[10946,17711,28657,46368,75025]
*Main> fibNth 3
2
*Main> fibNth 6
8
```

3 つの関数は美しく単純明快だ．我々は無限数列を定義するだけで，Haskell は作業に必要な部分だけを計算する．複数の無限数列を組み合わせることができるようになると本当に面白くなる．手始めに，開始値を 1 つだけずらした 2 つのフィボナッチ数列（の各項）を足し合わせてみよう．

```
*Main> take 5 (zipWith (+) fib (drop 1 fib))
[2,3,5,8,13]
```

なんと，やはりフィボナッチ数列ができてしまった．これらの高階関数は極めて相性が良い．上の例では，zipWith を呼んで，無限リストの（同じインデックスの）各項同士を綴じ合わせてペアを作り，それを + 関数に渡している．次のように書けば，範囲の各項を 2 倍できる．

```
*Main> take 5 (map (*2) [1 ..])
[2,4,6,8,10]
```

この例では map を用いて部分適用関数 `* 2` を無限範囲 `[1 ..]` に適用し，出来た無限範囲の先頭から 5 項を使っている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

関数型言語の良い所は，予想もしなかったような方法で組み合わせることができる点だ．例えば，関数合成は，部分適用関数および遅延数列と無理なく組み合わせることができる．

```
*Main> take 5 (map ((* 2) . (* 5)) fib)
[10,10,20,30,50]
```

このコードは盛りだくさんなので，分解して解説しよう．内側から順番に見ていく．まず，(* 5) だが，これは部分適用関数だ．この関数に何か渡すと，5 倍されて返ってくる．その結果を別の部分適用関数 (* 2) に渡す．こうしてできた合成関数を map に渡して，それをさらに無限フィボナッチ数列 fib の各項に適用する．最後に，この無限数列を take 5 に渡す．結果として，フィボナッチ数列の各項を 5 倍したあと，さらに 2 倍してできた数列の先頭の 5 項が返される．

こうして見ると，どのようにして解法を組み立てたかがよく分かる．単純に，ある関数を別の関数に渡しただけである．Haskell では，f . g x は f(g x) の略記法になっている．こんな具合に関数をいくつか作っていると，頭から順に適用したいと思うことがあるだろう．そんなときに使うのが . 演算子である．例えば，画像を反転させ，垂直軸の回りに対称移動させた後，水平軸の回りに対称移動させる場合，画像プロセッサは次のような処理を行う．

```
(flipHorizontally . flipVertically . invert) image
```

Simon Peyton-Jones 氏へのインタビュー

ここで一息入れるために，Haskell を開発した委員会の別のメンバーの話を聞いてみることにしよう．Simon Peyton Jones はロンドン大学ユニバーシティー・カレッジで 7 年間，グラスゴー大学で 9 年間教鞭をとった後，1998 年に，Microsoft Research（ケンブリッジ）に籍を移し，単一プロセッサマシンおよび並列マシン向けの関数型プログラミング言語の実装とアプリケーションを中心する研究に携わっている．彼は，本書で使ったコンパイラの主任デザイナーである．

Bruce Tate　Haskell を開発した経緯についてお聞かせいただけますか．

Simon Peyton-Jones　Haskell の非常に特殊な部分として，委員会による開発が成功を収めた言語であるという点が挙げられます．成功した言語を思い浮かべてみると，元々は特定の個人または極めて少人数で開発された言語であることが多いのですが，Haskell は違います．Haskell は，元々，20 人程度の研究者で構成される国際的なグループによって設計されました．Haskell は原則を重視する言語ですが，言語の中心となる原理原則については，メンバー間で十分に意志統一が図られていたため，設計の一貫性は維持されました．

また，Haskell は，設計されてから 20 年近く経った今，急激に人気が高まっています．プログラミング言語は，普通，開発後最初の数年で成功するか（ほぼ）失敗に終わるかが決まるものなのですが，今何が起こっているのでしょうか．Haskell の徹底した純粋主義，つまり副作用がないという特徴は，なじみのない制限だったため，これまで Haskell が主

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

流言語になるのを妨げていました．そういった長期的にメリットをもたらす特長が，やっと理解されるようになってきたのです．今後主流となる言語は，Haskell に似たものになるかどうかはともかく，副作用を抑える強力な仕組みを備えていると思います．

Bruce Tate　Haskell の機能で最も気に入っているのはどの部分ですか？

Simon Peyton-Jones　純粋さという点を除けば，Haskell の機能のなかで最も面白く普通とは異なる部分は，おそらく型システムでしょう．静的型は，現在利用可能なプログラム検証技術として，圧倒的な普及率を見せています．膨大な数のプログラマが毎日，型をコーディングしており（完全に型を書いているわけではないのですが），それらはプログラムをコンパイルするたびにコンパイラによってチェックされています．型というのは，関数型プログラミングの UML（統一モデル言語）です．つまり，プログラムの本質的かつ永続的な部分を形成する設計言語なのです．
Haskell の型システムは，型クラスや高度な型変数が用意されていたため，設計当初から，非常に表現力に富んでいました．当時から，Haskell は新しい型システムの概念を開拓する実験室のような役割を果たしており，私もこの「実験」を大いに楽しみました．マルチパラメータ型クラス，高ランク型，ファーストクラスとして扱えるポリモーフィズム，暗黙のパラメータ，GADT，型ファミリーなど，楽しみは尽きません．何より重要なのは，我々は，型システムによって静的にチェックできる特性の範囲を拡張しているのだという点です．

Bruce Tate　一から作り直せるとしたら，どの部分を変更しますか？

Simon Peyton-Jones　レコードシステムでしょうか．Haskell のレコードシステムが非常に単純化されているのにはいろいろと理由があるのですが，その単純さが弱点でもあります．
モジュールシステムも改良したいですね．具体的には，他の誰かに Haskell のパッケージ P を出荷するとき，「P を使うには，別の場所からインタフェース I と J をインポートする必要があります．これらを入手すれば，インタフェース K が提供されます」という風に指示できるようにしたいのです．Haskell には，このような指示を出すための正式な方法は用意されていません．

Bruce Tate　Haskell を用いたソリューションで，最も面白いと思ったものを教えてください．

Simon Peyton-Jones　Haskell は真の汎用プログラミング言語です．このことは長所であると同時に弱点でもあります．というのは，これといったキラーアプリケーションがないからです．その一方で，Haskell が，本当にエレガントで独特な問題解決方法を思いつくことができる媒体となることに気づくということはよくあります．例えば，Conal Elliot 氏の関数型のリアクティブプログラミングによるアニメーションに対する取り組みがあります．このアニメーションを見たおかげで，私は，「時間とともに変化する値は関数型プログラムによって操作可能な 1 つの値とみなすことができるのだ」という考え方に気づくことができました．もう少し日常的な（ですが非常に有用な）レベルでは，パー

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

サーやプリティプリント・コンビネータ（組み合わせ子）の豊富なライブラリがあります．これらのライブラリでは，簡単なインタフェースの背後に高度な知的レベルがカプセル化されています．もう 1 つ，Jean-Marc Eber 氏が金融デリバティブを記述する結合ライブラリの設計方法を見せてくれました．これは，私には思いもよらない使い方でした．どのケースも，Haskell という媒体によって，主流言語でははるかに実現が難しかったであろうと思われる新しいレベルの表現が可能になっています．

ここまで読み終えた皆さんは，Haskell でかなり高度な問題に挑戦できるだけの知識を習得しているはずだ．が，その一方で，I/O，状態，エラー処理など，簡単なことがまだできない．これらの問題を解決するには高度な理論を導入することになる．3 日目では，モナドについて解説する．

2 日目で学んだこと

2 日目は，高階関数について見てきた．本書に登場した他の大半の言語に出てきたのと同種のリストライブラリについて説明した．`map`，いくつかのバージョンの `fold`，および `zip` や `zipWith` などの関数を紹介した．固定長のリストの操作について見たあと，Clojure で用いたのと同じ遅延評価についても解説した．

高度な機能について説明する際，関数に一部の引数だけを与える手法を学んだ．部分適用関数という手法だ．その後，部分適用関数を用いて，一度に複数の引数をとる関数（`f(x, y)`）を，一度に 1 つずつ引数をとる関数（`f(x)(y)`）に変換した．Haskell では，すべての関数がカリー化されることも学んだ．この事実により，複数の引数をとる Haskell の関数の型シグニチャの説明がつく．例えば，関数 `f x y = x + y` の型シグニチャは `f :: (Num a) => a -> a -> a` である．

関数合成（ある関数が返す結果を別の関数の入力として用いる手法）についても学んだ．関数合成を使うと，関数を数珠つなぎにすることができる．

最後に，遅延評価を取り上げた．無限リストを作る関数を定義できた．その無限リストを必要なときに必要な部分だけ使えばよい．この方法でフィボナッチ数列を定義した．また，関数合成と遅延数列を組み合わせて，新しい遅延数列を無理なく作成できた．

セルフスタディ 2 日目

探してみよう：

- リスト，文字列，またはタプルに使用できる関数
- リストをソートする方法

試してみよう：

- リストを受け取り，ソートされたリストを返すソートプログラムを書け．
- リストと，2 つの引数を比較する関数を受け取り，ソートされたリストを返すソートプログラムを書け．
- 文字列を数値に変換する Haskell の関数を書け．文字列の形式は`$2,345,678.99`とし，先頭に 0 がくる可能性もあるものとする．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

- x を引数にとり，遅延シーケンスとして，x で始まる 2 つ飛びの数列を返す関数を書け．次に，y を引数にとり，y で始まる 4 つ飛びの数列を含む関数を書け．これらの関数をリストの作成構文で組み合わせて，x + y で始まる 8 つ飛びの数列を返す関数を作成せよ．
- 部分適用関数を用いて，ある数値の 2 分の 1 を返す関数と文字列の末尾に\n を追加する関数を定義せよ．

もっと面白いものをご所望の方に，もう少し歯ごたえのある問題を出そう．

- 2 つの整数の最大公約数を求める関数を書け．
- 素数の遅延シーケンスを作成せよ．
- 長い文字列を適切な単語境界で複数の行に分割せよ．
- 上の問題で，分割した行に行番号を付けよ．
- 上記の各問題で，テキストをスペースで左寄せ，右寄せ，中央寄せ（左右の余白を揃える）にする関数を追加せよ．

8.4　3 日目：精神融合

『スター・トレック』では，スポックは，お互いの意識を共有する特殊能力精神融合を備えていた．Haskell の支持者のなかには，Haskell と自分自身との間に，まさにそのような結びつきを感じると主張する人が多い．多くの支持者の間で最も敬意を集めている Haskell の機能は，その型システムだ．Haskell の学習にかなりの時間を費やした筆者には，その理由が容易に分かる．Haskell の型システムは柔軟で豊かなため，プログラマの意図をほぼすべて推論してくれる．しかも，必要なとき以外，プログラマが型システムを意識することはない．また，型は，関数を組み立てるときのサニティチェックにもなる（とりわけ抽象的な関数合成のときには重要だ）．

クラスと型

Haskell の型システムは強力な機能である．型推論があるため，プログラマにとって過度の負担はない．また，微妙なプログラミングエラーでさえ捕捉するくらいに堅牢でもある．さらには，多相的（ポリモーフィック）である．すなわち，見かけの違う同じ型を同じものとして扱うことができる．ここでは，型の例をいくつか紹介した後，我々独自の型を作ってみる．

基本型

これまでに登場したいくつかの基本型について復習しておこう．まず，シェルで型オプション（t オプション）をオンにする．

```
Prelude> :set +t
```

これで，各文が返す型を確認できる．いくつか文字と文字列で試してみよう．

```
Prelude> 'c'
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
'c'
it :: Char
Prelude> "abc"
"abc"
it :: [Char]
Prelude> ['a', 'b', 'c']
"abc"
it :: [Char]
```

it は常に最後に入力したものの値を示し，::は「その型は？」と読める．Haskell では文字はプリミティブ型だ．文字列は文字の列である．文字の列の表現には配列を用いてもダブルクオートを用いても構わない．Haskell から見ると，どちらでも値は同じである．

```
Prelude> "abc" == ['a', 'b', 'c']
True
```

以下に示すとおり，プリミティブ型はほかにもいくつかある．

```
Prelude> True
True
it :: Bool
Prelude> False
False
it :: Bool
```

これから型付けについて詳しく見ていくが，上記の説明を頭に入れておけば何が起こっているのか分かるだろう．それでは，我々独自の型をいくつか定義してみよう．

ユーザ定義の型

ユーザ独自のデータ型を定義するには data キーワードを使う．最も簡単な型宣言では，有限個の値を列挙する．例えばブール型は次のように定義される．

```
data Boolean = True | False
```

これは，Boolean 型は単一値，すなわち True または False をとるという意味だ．同じ要領でユーザ独自の型を定義できる．2 種類の組（スーツ）と 5 つの数（ランク）からなる簡素化されたトランプを考えてみる．

▶ code/haskell/cards.hs

```
module Main where
    data Suit = Spades | Hearts
    data Rank = Ten | Jack | Queen | King | Ace
```

この例では，Suit と Rank が型構成子だ．data を用いて，新しいユーザ定義の型を作成している．次のようにして，このモジュールを読み込む．

```
*Main> :load cards.hs
[1 of 1] Compiling Main             ( cards.hs, interpreted )
Ok, modules loaded: Main.
*Main> Hearts

<interactive>:1:0:
    No instance for (Show Suit)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

```
        arising from a use of 'print' at <interactive>:1:0-5
```

うわっ．何があったのだろう？ このエラーメッセージは要するに「コンソールがこれらの値を表示しようとしたが，その方法が分からない」と言っている．ユーザ定義のデータ型を宣言するとき，show 関数を利用できるようにする略記法がある．次のように指定する．

▶ code/haskell/cards-with-show.hs

```
module Main where
    data Suit = Spades | Hearts deriving (Show)
    data Rank = Ten | Jack | Queen | King | Ace deriving (Show)
    type Card = (Rank, Suit)
    type Hand = [Card]
```

いくつか型の別名を追加している点に注意してほしい．Card は組（スーツ）と数（ランク）のタプルだ．Hand はカードのリストだ．これらの型を用いて，次のような新しい関数を作成する．

▶ code/haskell/cards-with-show.hs

```
value :: Rank -> Integer
value Ten = 1
value Jack = 2
value Queen = 3
value King = 4
value Ace = 5

cardValue :: Card -> Integer
cardValue (rank, suit) = value rank
```

どのようなカードゲームでもそうだが，カードの数（ランク）を割り当てることができなければならない．組（スーツ）はここでは関係ない．そこで，Rank の value を計算する関数と cardValue を計算する関数を定義した．では実行してみよう．

```
*Main> :load cards-with-show.hs
[1 of 1] Compiling Main             ( cards-with-show.hs, interpreted )
Ok, modules loaded: Main.
*Main> cardValue (Ten, Hearts)
1
```

ユーザ定義の型の複雑なタプルを使っている．型システムがこちらの意図を明確に表しているので，処理内容を容易に推測できる．

関数とポリモーフィズム

先ほど，いくつか関数型が出てきた．簡単な関数で見てみよう．

```
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

この関数に型を追加してみる．次のようになる．

```
backwards :: Hand -> Hand
...
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:32:48 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

これだと，backwards関数で操作できる対象が，1種類のリスト，つまりcardのリストだけに限定される．本当は次のようにしたい．

```
backwards :: [a] -> [a]
backwards [] = []
backwards (h:t) = backwards t ++ [h]
```

これでbackwards関数はポリモーフィック（多相的）になった．[a]は，任意の型のリストを使えるという意味だ．つまり，型aのリストを受け取り，同じ型aのリストを返す関数を定義できる．[a] -> [a]と書くことによって，この関数で使える型のテンプレートを作成している．これで，最初がIntegerのリストなら返されるのもIntegerのリストであるとコンパイラに伝えたことになる．型をごまかせないだけの情報をHaskellに提供したことになるのである．

ポリモーフィックなデータ型を作ってみよう．以下に，同じ型の3つの点からなる3要素のタプルを作る．

▶ code/haskell/triplet.hs

```
module Main where
    data Triplet a = Trio a a a deriving (Show)
```

左辺はdata Triplet aとなっている．この例ではaは型変数だ．これにより，同じ型の要素を持つ3要素のタプルはTriplet aという型で表現できるようになる．試してみよう．

```
*Main> :load triplet.hs
[1 of 1] Compiling Main             ( triplet.hs, interpreted )
Ok, modules loaded: Main.
*Main> :t Trio 'a' 'b' 'c'
Trio 'a' 'b' 'c' :: Triplet Char
```

上の例では，データ構成子Trioを使って3要素のタプルを作成している．データ構成子については，次のセクションで詳しく説明する．型宣言のとおり，結果はTriplet a，より厳密にはTriplet charとなっており，Triplet aを要求する任意の関数で使える．上の例は，同じ型の3つの要素からなる型を記述する，型のテンプレート全体を作成している．

再帰型

再帰的な型を定義することもできる．例えばツリーを考えてほしい．方法はいくつかあるが，ここで作成するツリーでは，リーフノードに値が格納される．ノードは，リーフまたはツリーのリストのどちらかである．このツリーは，次のように記述できる．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/tree.hs

```
module Main where
    data Tree a = Children [Tree a] | Leaf a deriving (Show)
```

この定義では，1 つの型構成子 Tree と 2 つのデータ構成子 Children と Leaf を使っている．次のように，これらすべてを使ってツリーを表現できる．

```
Prelude> :load tree.hs
[1 of 1] Compiling Main             ( tree.hs, interpreted )
Ok, modules loaded: Main.
*Main> let leaf = Leaf 1
*Main> leaf
Leaf 1
```

手始めに，1 つのリーフを持つツリーを作成する．新しいリーフは変数に代入する．データ構成子 Leaf の唯一の仕事は，型と値を一緒に保持することだ．各値を取り出すには次のようにパターンマッチングを用いる．

```
*Main> let (Leaf value) = leaf
*Main> value
1
```

もう少し複雑なツリーを作成してみよう．

```
*Main> Children[Leaf 1, Leaf 2]
Children [Leaf 1,Leaf 2]
*Main> let tree = Children[Leaf 1, Children [Leaf 2, Leaf 3]]
*Main> tree
Children [Leaf 1,Children [Leaf 2,Leaf 3]]
```

どちらもリーフである 2 つの子ノードを持つツリーを作成する．次に，左側がリーフ，右側がツリーであるような 2 つのノードを持つツリーを作成する．ここでも，パターンマッチングを使って各ノード値を取り出すことができる．さらに複雑にしてみよう．この定義は再帰的なので，let とパターンマッチングを使って，いくらでも深いツリーを作成できる．

```
*Main> let (Children ch) = tree
*Main> ch
[Leaf 1,Children [Leaf 2,Leaf 3]]
*Main> let (fst:tail) = ch
*Main> fst
Leaf 1
```

設計の意図は型から明解であり，作業に必要な部分はマッチングで引き剥がせる．この設計戦略には明らかにオーバーヘッドが伴うが，抽象化の度合いが深くなってくると，そうした余分なオーバーヘッドという代価を支払うだけの価値が出てくることがある．この例では，型システムによって，個々の型構成子に関数を結びつけることができる．ツリーの深さを調べる関数を作成してみよう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/tree.hs

```
depth (Leaf _) = 1
depth (Children c) = 1 + maximum (map depth c)
```

この関数の最初のパターンはシンプルだ．ツリーが1つのリーフである場合は，そのリーフの中身に関係なく，ツリーの深さは1だ．

次のパターンはもう少し複雑だ．Children に対して depth を呼び出す場合は，maximum (map depth c) に 1 を足す．maximum 関数はリスト中の最大要素を返す．map depth c については，前にも見たように，すべての子ノードの深さのリストを返す．作業に必要なデータ構造の一部をマッチさせるのにデータ構成子を使う方法が，この例で分かるはずだ．

クラス

ここまで，型システムとその動作をいくつか例を挙げて見てきた．ユーザ定義の型構成子を作成し，型のテンプレートを用意した．また，型のテンプレートによって，データ型を定義し，それらのデータ型を使う関数を宣言した．実は Haskell には，型に関してもう1つ重要な概念がある．しかもその重要度はかなり大きい．それはクラスと呼ばれる概念だ．と言っても，オブジェクト指向のクラスではないので注意してほしい．このクラスにはデータは関係ない．Haskell では多相性をうまく制御し，クラスを使って多重定義型を実現している．

例えば，2つのブール値を足し算することはできないが，2つの数値なら足し算できる．Haskell では，この判別を行うためにクラスを用いる．具体的にいうと，クラスとは，入力に応じて，どの演算が実行可能かを定義したものである．Clojure のプロトコルと同じだと考えてよい．

以下に，クラスの動作方法を解説する．クラスには，いくつかの関数シグニチャが定義されている．ある型がこれらすべての関数をサポートしていれば，その型は当該クラスのインスタンスである．例えば，Haskell のライブラリには，Eq というクラスがある．

このクラスの定義は次のようになる．

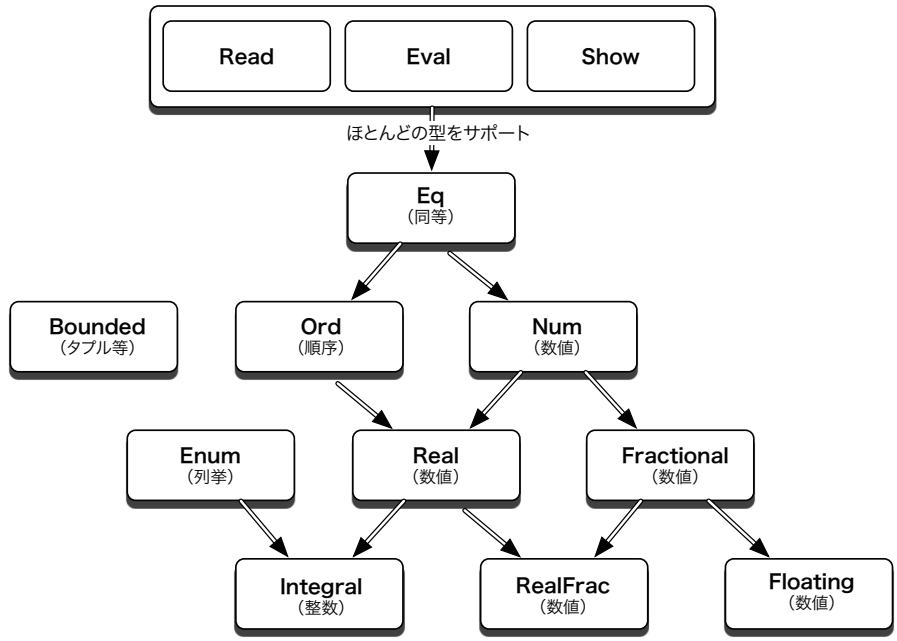
```
class  Eq a  where
   (==), (/=) :: a -> a -> Bool

       -- Minimal complete definition:
       --       (==) or (/=)
   x /= y     =  not (x == y)
   x == y     =  not (x /= y)
```

したがって，ある型が==と/=の両方をサポートしていれば，その型は Eq のインスタンスである．お決まりの実装を指定することもできる．また，あるインスタンスで上記のどちらか一方の関数を定義すれば，他方は何もしなくても使用可能となる．

型クラスは継承をサポートしており，継承と聞いて期待するとおりの動作をする．例えば，Num クラスには，Fractional と Real の2つのサブクラスがある．Haskell 98 の大半の主要クラスの階層を図 8.1「Haskell の主要クラス」(次ページ)に示す．これらのクラスのインスタンスは型であってデータオブジェクトではない点に注意してほしい．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社



▲図 8.1　Haskell の主要クラス

モナド

本書の執筆を決断したときから，筆者は，モナドに関するセクションを書くのがとても恐かった．しかし，いくらか勉強してみると，この概念はまったく難しくないことが分かってきた．このセクションでは，モナドが必要となる理由を直感的に順を追って説明する．その後，モナドの構築方法について概説する．最後に，モナドの動作方法がよく理解できるシンタックスシュガーを紹介する．

筆者自身，モナドの理解の助けとなるいくつかのチュートリアルで学習した．Haskell の wiki[*3]には，良いサンプルが掲載されており，筆者も参考にした．また，Understanding Monads[*4]という記事にも，実用的な良い例が掲載されている．しかし，モナドでできることを理解するには，結局さまざまなリソースのたくさんの例に目を通す必要があることが分かるだろう．

[*3] http://www.haskell.org/tutorial/monads.html
[*4] http://en.wikibooks.org/wiki/Haskell/Understanding_monads

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

問題：酔った海賊

ある海賊が宝探しの地図を作成しているとする．この海賊は酔っているため，既知のポイントと既知の方向を選択し，宝のありかまでよろめきはいつくばって進んでいく．1回よろめくと2歩進み，1回腹ばいになると1歩進む．命令型言語では，次のように文を一続きに並べていく．ここで，vはスタート地点からの距離を保持する値だ．

```
def treasure_map(v)
    v = stagger(v)
    v = stagger(v)
    v = crawl(v)
    return( v )
end
```

treasure_map内ではいくつかの関数を呼び出す．これらの関数を呼び出すと，状態，すなわち移動した距離が逐次変化していく．問題は，可変状態が含まれる点だ．関数型プログラミングでは，同じ問題を次のようにコーディングする．

▶ code/haskell/drunken-pirate.hs

```
module Main where

    stagger :: (Num t) => t -> t
    stagger d = d + 2
    crawl d = d + 1

    treasureMap d =
        crawl (
        stagger (
        stagger d))
```

見てのとおり，この関数定義は読みづらい．stagger，stagger，crawlの順ではなく，crawl，stagger，staggerの順になっており，引数の位置もぎこちない．このような分かりにくい定義ではなく，複数の関数を順番に数珠つなぎにできる方法を使いたい．それには，次のようにlet式を用いる．

▶ code/haskell/drunken-pirate.hs

```
letTreasureMap (v, d) = let d1 = stagger d
                            d2 = stagger d1
                            d3 = crawl d2
                        in d3
```

Haskellでは，let式を数珠つなぎにし，最終的な形をin文で表現できる．しかし，この定義も，不満が残るという点では最初の定義とほとんど変わらない．入出力の型がどれも同じなのだから，この類の関数たちを簡単に組み合わせる方法があってもよさそうなものだ．stagger(crawl(x))を，stagger(x) → crawl(x)のような形で書けるとうれしい．これこそがモナドである．

モナドを一言で表すと，特別なやり方で複数の関数を組み合わせるための方法だと言える．Haskellでは，さまざまな目的にモナドを使う．純粋な関数型言語では，関数は同じ入力を与えられたとき同じ結果を生成しなければならないので，I/Oのようなものを扱

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

うのは難しい．I/O では，例えば，ファイルの中身の状態に応じて関数の処理内容も変えることが求められるからだ．

また，先ほどの酔った海賊のようなコードは状態を保持しているから動作する．モナドを使うと，プログラムの状態をシミュレートできる．Haskell では，*do* 記法という特別な構文を用意している．これにより，命令形式でプログラムを書くことができる．do 記法はモナドに依存している．

最後に，Haskell では，エラー条件のような非常に簡単なことが難しい．というのは，関数が正常終了したかどうかによって，返されるモノの型が異なるからだ．これを解決するため，Haskell では `Maybe` モナドを用意している．もう少し掘り下げてみよう．

モナドの構成要素

モナドは，基本的なレベルでは，次の 3 つの基本要素で構成される．

- コンテナとなるものの型を変数に取る型構成子．コンテナとして使えるのは，単純な変数やリストなど，値を保持できるものならなんでもよい．このコンテナに関数を収めることになる．どのコンテナを選ぶかは，モナドで実行したいことによって異なる．
- 関数をラップしてコンテナに格納する `return` という名前の関数．この名前は，後で do 記法について説明するときに分かるだろう．ここでは，`return` は関数をラップしてモナドに入れ込むという点だけ覚えておいてほしい．
- 関数をラップする`>>=`（バインド）という名前の関数．バインドを使って，関数を数珠つなぎにする．

すべてのモナドは 3 つのルールを満たす必要がある．ここでそれらのルールについて簡単に触れておく．モナド `m`，関数 `f`，値 `x` について，次のルールがある．

- ラップした値はそのまま関数に渡すことができなければならない（`return x >>= f = f x`）．
- 情報を失うことなく値をアンラップおよびラップできなければならない（`monad >>= return = monad`）．
- バインドをネストした場合と，それらをシーケンシャルに呼び出した場合とで結果が同じでなければならない（`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`）．

これらのルールについて詳細な解説は行わないが，これらが必要な理由は明らかだ．これらのルールに従うことによって，情報を失うことなく，有用な変換が可能となる．詳しく知りたい方は文献をあたってほしい．

理論はこのくらいにして，さっそく簡単なモナドを作成してみよう．ここではモナドを一から作成し，最後にいくつか有用なモナドを紹介して章を締めくくることにする．

モナドを一から作成する

まず必要なのは型構成子だ．このモナドは，次のように 1 つの関数と 1 つの値を持つ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/drunken-monad.hs

```
module Main where
    data Position t = Position t deriving (Show)

    stagger (Position d) = Position (d + 2)
    crawl (Position d) = Position (d + 1)

    rtn x = x
    x >>== f = f x
```

モナドの3つの主要な要素は，型構成子，return，バインドだ．このモナドは，考え得る限り最もシンプルなものだ．型コンテナは，シンプルな型構成子 data Position t = Position t だ．ここでは単に，任意の型のテンプレートに基づいて基本型を定義しているだけだ．次に関数を値としてラップする return が必要だ．このモナドは非常にシンプルなので，モナド自身の値を返すだけでよい．その際，rtn x = x によって正しくラップしている．最後に，関数を合成するためのバインドが必要だ．このモナドのバインドは>>==と呼ばれ，対応する関数をモナド内の値を引数として呼び出しているだけだ(x >>== f = f x)．ここでは，Haskell の組み込みのモナド関数と衝突するのを避けるために，>>=と return の代わりに>>==と rtn を使っている．

stagger と crawl を，裸の integer ではなく，この自家製のモナドを使うように書き換えている点にも注意してほしい．では，このモナドをテストしてみよう．思い出してほしい．我々が求めていたのは，関数の入れ子を関数の合成に変換するシンタックスだった．修正版の宝探しの地図は次のようになる．

▶ code/haskell/drunken-monad.hs

```
treasureMap pos = pos >>==
                  stagger >>==
                  stagger >>==
                  crawl >>==
                  rtn
```

実行してみると，次のように期待どおりに動く．

```
*Main> treasureMap (Position 0)
Position 5
```

モナドと do 表記

シンタックスは随分良くなったが，さらに良くするためのシンタックスシュガーも容易に想像がつく．Haskell の do 記法がまさにそのシンタックスシュガーだ．do 記法は，特に I/O などの問題を解くときに便利だ．次のコードは，do 記法を用いて，コンソールから1行読み込み，その行を逆順にして出力するものだ．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/haskell/io.hs

```
module Main where
    tryIo = do  putStr "Enter your name: " ;
                line <- getLine ;
                let { backwards = reverse line } ;
                return ("Hello. Your name backwards is " ++ backwards)
```

このプログラムの先頭は関数の宣言である点に注目してほしい．その後の簡単な do 表記が，モナドをくるむシンタックスシュガーになっている．このシンタックスシュガーによって，このプログラムはステートフルな命令型であるように感じられるが，実際にはモナドを使っている．次の構文ルールを知っておくとよい．

代入には<-を使う．GHCi では，文をセミコロンで区切り，do 式の本体を含める．その中に let 式を中括弧で囲んで指定する．複数の行がある場合は，コードを:{と}:でラップして，それぞれを別の行に書く．さて，ここでモナドのラッピング構文を return と呼ぶ理由が明らかになる．この構文は，関数が返す結果を，do 表記で吸収できる整理された形式できれいにパッケージ化する．このコードはステートフルな命令側言語で書いたように動作するが，実際にはモナドを使ってステートフルなやり取りを管理している．すべての I/O はしっかりとカプセル化されており，do ブロック内の I/O モナドの 1 つを使用して捕捉する必要がある．

さまざまな計算戦略

すべてのモナドには計算戦略が関連付けられている．酔った海賊の例で用いた identity（恒等）モナドは，入力したものをそのまま返すものだった．我々はこのモナドを使って，入れ子になったプログラム構造を逐次化された平坦なプログラム構造に変換した．もう 1 つ別の例を見てみよう．奇妙に思えるかもしれないが，リストもモナドなのだ．その return とバインド（>>=）は次のように定義されている．

```
instance Monad [] where
    m >>= f  = concatMap f m
    return x = [x]
```

モナドには，コンテナと型構成子，関数をラップする return メソッド，関数をアンラップするバインドメソッドが必要なことを思い出してほしい．モナドはクラスであり，[] はそのクラスのインスタンスであり型構成子である．次に，結果を return としてラップする関数が必要になる．

リストの場合，関数をリスト内にラップする．アンラップするには，バインドが map を用いてリストの各要素に対して関数を呼び出し，その結果を連結する．concat と map は順番に適用されることが頻繁にあるため，両方を実行する便利な関数が用意されているが，単純に concat (map f m) と書いても構わない．

リストモナドが動作する様子の感じをつかんでもらうために，do 表記で書かれた次のスクリプトを見てみよう．

```
Main> let cartesian (xs,ys) = do x <- xs; y <- ys; return (x,y)
Main> cartesian ([1..2], [3..4])
[(1,3),(1,4),(2,3),(2,4)]
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

do記法とモナドを使って簡単な関数を作成した．xsのリストからxを，ysのリストからyを取り出し，xとyの各組み合わせを返す．ここからパスワードクラッカーを作るのは簡単だ．

▶ code/haskell/password.hs

```
module Main where
    crack = do x <- ['a'..'c'] ; y <- ['a'..'c'] ; z <- ['a'..'c'] ;
               let { password = [x, y, z] } ;
               if attempt password
                    then return (password, True)
                    else return (password, False)

    attempt pw = if pw == "cab" then True else False
```

この例では，リストモナドを使って考えられるすべての組み合わせを計算している．このコンテキストでは，x <- [lst]は，「[lst]から取り出した各xについて」という意味だ．Haskellに力仕事をさせている．ここまでくれば，あとは作成した各パスワードを試すだけだ．本当のパスワードはattempt関数内にハードコーディングしてある．この問題を解くために使える計算戦略はほかにもたくさん考えられるが（例えばリスト内容表記など），この問題は，リストモナドの背後にある計算戦略を示してくれる．

Maybeモナド

ここまで，Identityモナドと Listモナドについて見てきた．後者については，モナドが中心をなす計算戦略をサポートしていることを学んだ．このセクションでは，Maybeモナドについて解説する．ここではMaybeモナドを使って一般的なプログラミングの問題を処理する．すなわち，関数によってはエラーになる可能性があるという問題だ．というと，データベースや通信の分野の話をしていると思うかもしれないが，それよりはるかに簡単なAPIでもエラーという概念をサポートしなければならないことが多い．文字列のインデックスを返すような文字列検索を考えてみてほしい．文字列が存在するなら結果の型はIntegerで，さもなければNothingだ．

こんな計算をちまちまつなぎ合わせるのは退屈だ．Webページを解析する関数があるとしよう．HTMLページ，ページ内のボディ部，そのボディ内の最初の段落を取得したいとする．次のようなシグネチャを持つ関数をコーディングする．

```
paragraph :: XmlDoc -> XmlDoc
...

body :: XmlDoc -> XmlDoc
...

html :: XmlDoc -> XmlDoc
...
```

これらのシグネチャから，次のような関数を書くことができる．

```
paragraph body (html doc)
```

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

問題は，paragraph，body，および html の各関数はエラーになる可能性があるため，Nothing となる型を許す必要があるという点だ．Haskell はそのような型として Just が用意されている．Just x を使うと，Nothing，または特定の型を次のようにラップできる．

```
Prelude> Just "some string"
Just "some string"
Prelude> Just Nothing
Just Nothing
```

Just は，パターンマッチングを用いて除去できる．先ほどの例に戻ると，paragraph，body，html の各ドキュメントは，Just XmlDoc を返す可能性がある．あとは，Haskell の case 文（Erlang の case 文と同じ）とパターンマッチングを使って，次のように書くことができる．

```
case (html doc) of
  Nothing -> Nothing
  Just x  -> case body x of
               Nothing -> Nothing
               Just y  -> paragraph 2 y
```

しかし，この結果はまったく満足のいくものではない．我々は，paragraph 2 body (html doc) という様なコードが書きたかったのだから．我々が本当に必要としているのは Maybe モナドだ．次のその定義を示す．

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return          = Just
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x

    ...
```

ここでラッピングしている型は，Maybe a という型構成子だ．この型は，Nothing または Just a をラップできる．

return は簡単だ．結果を Just でラップしているだけだ．バインドも簡単だ．Nothing の場合は，Nothing を返す関数を返す．Just x の場合は，x を返す関数を返す．どちらも return でラップされる．これで，これらの操作を簡単につなげることができる．

```
Just someWebPage >>= html >>= body >>= paragraph >>= return
```

以上で，要素を完全に組み合わせることができる．これが機能するのは，我々が合成する関数を介した意志決定をモナドが代行してくれるからだ．

3 日目で学んだこと

3 日目は，Haskell の 3 つの難しい概念に取り組んだ．すなわち，型，クラス，モナドの 3 つだ．まず，既存の関数，数値，ブール値，文字から推論されるいくつかの基本的な型について見た．次に，ユーザ定義の型をいくつか定義した．基本的な例として，型を用

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:36:06 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

いて，一部の組（スーツ）と数（ランク）からなる簡素化されたトランプを定義した．型をパラメータ化する方法，および再帰型定義を使う方法についても学んだ．

次に，本章の締めくくりとしてモナドについて解説した．Haskell は純粋な関数型言語なので，命令型形式で問題を表現したり，プログラムの実行に伴って状態を蓄積したりする処理が難しくなる．モナドは，関数をラップして数珠つなぎにする型構成子だ．異なる型コンテナを使ったモナドを組み合わせれば，異なる種類の計算戦略を実現できる．モナドを使って，我々のプログラムをより自然な命令型方式に変換し，モナドの具体的な実例をいくつか紹介した．

セルフスタディ 3 日目

探してみよう：

- いくつかのモナドのチュートリアル
- Haskell 組み込みのモナドの一覧

試してみよう：

- ハッシュテーブルの値を検索する関数を `Maybe` モナドを用いて書け．他のハッシュを格納するハッシュを数レベルの深さまで書け．`Maybe` モナドを使って，ハッシュキーに対応する要素を数レベルの深さまで検索せよ．
- Haskell で迷路を作成してみよ．`Maze` 型と `Node` 型，および条件が与えられたときにノードを返す関数が必要になる．ノードは他のノードへの出口のリストを持つ必要がある．
- List モナドを用いて，上の迷路を解いてみよ．
- モナドを非関数型言語で実装してみよ（Ruby でモナドを実装するための連載記事を参照されたい[*5]）．

8.5 Haskell のまとめ

本書に登場したすべての言語のなかで，（一個人ではなく）委員会によって開発されたのは Haskell だけだ．遅延評価セマンティクスを備えた純粋な関数型言語が複数普及してきたため，既存の機能と今後の研究を統合するオープンな標準を構築するための委員会が結成された．このようにして Haskell が誕生し，1990 年にバージョン 1.0 が開発された．以来，Haskell とそのコミュニティは成長を遂げてきた．

Haskell は，リスト内包表記，遅延コンピューティング戦略，部分適用関数，カリー化といった広範な関数型機能をサポートしている．実際，デフォルトの設定では，Haskell の関数は 1 度に 1 つのパラメータしか処理しないが，カリー化を用いて複数の引数をサポートしている．

Haskell の型システムは，型の安全性と柔軟性のバランスがよくとれている．完全に多

[*5] `http://moonbase.rydia.net/mental/writings/programming/monads-in-ruby/00introduction.html`

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

相的なテンプレートシステムによって，ユーザ定義の型および型クラス（インタフェースの継承を完全にサポート）を巧妙に実現している．通常，Haskell のプログラマは関数を宣言するときを除き，型の詳細について意識する必要はない．型システムがあらゆる種類のエラーからプログラムを保護してくれるからだ．

純粋な関数型言語はどれもそうだが，Haskell でも，開発者は命令型プログラムと状態の蓄積の処理には工夫を凝らす必要がある．I/O 処理も難しい．幸いにも，Haskell プログラマは，こうした問題の解決にモナドを使うことができる．モナドは，型構成子であり，関数を値としてラップおよびアンラップする基本的な関数をサポートするコンテナでもある．コンテナの型が異なると計算戦略も異なってくる．これらの関数を使うと，プログラマは *do* 記法によってモナドを自由に数珠つなぎにできる．このシンタックスシュガーによって，制限はあるものの，命令型プログラムを書くことができる．

長所

Haskell は，妥協なしの徹底した純粋な関数型アプローチを採用しているため，利点と欠点が極端な形で現れることが多い．一つひとつ見ていこう．

型システム

強い型付けが好みであれば（いや，そうでない人もおそらく）Haskell の型システムを気に入るだろう．この型システムは必要なときに役に立つと同時に，不要なときに出しゃばったりしない．よくあるエラーからの保護における有用性が向上するため，実行時ではなく，コンパイル時にエラーを捕捉できる．しかしこうした安全性の向上は，型システムがもたらす利点の一部にすぎない．

Haskell の型に関して最も興味深い部分はおそらく，新しい型を新しい動作に非常に容易に関連付けることができる点だろう．ユーザは高度な型を一から作り上げることができる．型構成子とクラスを用いて，モナドなどの極めて複雑な型とクラスをたやすくカスタマイズすることもできる．クラスを使うと，新しいカスタム型の定義に既存の Haskell ライブラリを利用できる．

表現性

Haskell は素晴らしいパワーを備えた言語だ．抽象的な意味では，強力な概念をコンパクトに表現するために必要なものすべてを備えている．これらの概念は，豊富な関数ライブラリと強力なシンタックスを介して振る舞いを包み込むものである．これらの概念を拡張することにより，型を作成するデータ型や，過剰なシンタックスを使わずに，正しいデータに正しい関数をバインドする再帰型へとつながる．研究教育分野では，関数型プログラミングを教えるのに Haskell ほど強力な言語はほかにない．Haskell には，関数型プログラミングに必要なものがすべて揃っている．

プログラミングモデルの純粋さ

純粋なプログラミングモデルを採用すると，問題へのアプローチの方法が根本的に変わる可能性がある．従来の古いプログラミングパラダイムから脱却して，別のやり方を全面的に受け入れることを余儀なくされる．純粋な関数型原語には，ある種の信頼がお

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ける．関数は，同じ入力が与えられれば，必ず同じ出力を返す．この性質により，プログラムの推論が極めて容易になる．場合によっては，プログラムが正しいかどうかを証明することも可能だ．また例えば，並行処理において予想外に処理が複雑化したり，動作が不安定または低速になるといった，副作用に依存していることに起因する多くの問題から解放される可能性がある．

遅延セマンティクス

かつては関数型言語での処理といえば再帰処理を意味した．遅延計算は，データを処理するためのまったく新しい戦略を提供するものだ．この戦略により，パフォーマンスの高いプログラムを構築できる．また必要なコード行数も他の戦略の数分の 1 で済むことが多い．

学界のサポート

極めて重要で影響力のある言語のなかには，例えば Pascal など，学究的な世界で成長してきたものがある．こうした言語は，学問分野で研究され使われることで大きく進歩してきた．Haskell は，最も広く使われている関数型プログラミング手法の教材用言語として，改良と成長が続いている．Haskell は完全な主流言語ではないものの，重要な仕事を実行するのに必要な数人程度のプログラマはいつでも見つかるようになるだろう．

短所

もうお分かりだと思うが，すべての仕事を完璧にこなすプログラミング言語など存在しない．Haskell の長所の多くは，そのまま短所でもある．

プログラミングモデルの柔軟性の欠如

純粋な関数型言語には利点もあるが，同時にいくつか問題点も抱えている．気づいた方もいるかもしれないが，本書では，最後の章の一番最後にモナドによるプログラミングを取り上げた．これはある意味，当然だ．モナドの概念を理解するには高い知性が求められる．しかし，本書では，他の言語では取るに足らないことを実行するのにモナドを使った．具体的には，命令型プログラムの作成，I/O 処理，さらには価値があるのかないのか分からないリスト関数の処理にまでモナドを使ってみた．他の言語についても書いたことがあるが，ここでもう一度書いておく．Haskell では，難しいことが簡単になる場合と，簡単なことが難しくなる場合とがある．

特定のスタイルは特定のプログラミングパラダイムに有用である．手順を追ったアルゴリズムを構築する場合は，命令型言語が向いている．大量の I/O 処理やスクリプティングには，関数型言語は向いていない．ある人の目には純粋に見えることも，別の人の目には妥協がない（融通が利かない）ように見えることがある．

コミュニティ

妥協といえば，Scala と Haskell でアプローチの方法に大きな違いが見てとれる．どちらも強く型付けされた言語だが，その哲学は根本的に異なる．Scala は妥協を求めた言語であり，Haskell は純粋さを求めた言語だ．Scala は妥協することによって，当初 Haskell

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

よりも大きなコミュニティを形成した．プログラミングコミュニティの大きさによって言語の成功の度合いを測ることはできないが，成功するには十分な規模のコミュニティが必要だし，ユーザの数が増えればその言語を使う機会も多くなり，コミュニティの資源も増える．

学習曲線

Haskell の概念で，理解するのに高い知性を必要とするのは，モナドだけではない．複数の引数をとるすべての関数はカリー化される．大半の基本的な関数はパラメータ化された型を持ち，数値を操作する関数は型クラスを使用することが多い．こうした概念を習得することによって得られる報酬は最終的にはその価値が十分にあるかもしれないが，しっかりとした理論的基礎を備えた強いプログラマでなければ Haskell で成功するチャンスはない．

終わりに

本書で取り上げた関数型言語のなかで，Haskell は最も難しかった．モナドと型システムが重視されているため，学ばなければならないことが多い．しかし，いったん主要な概念を習得してしまえば，あとは楽なので，筆者が学習した言語のなかでは最も学習し甲斐のある言語となった．型システムとモナド適用のエレガントさを考慮すると，いつか，この言語を本書の中でも最も重要な言語のひとつとして振り返る日がくるだろう．

Haskell は別の役割も果たしている．そのアプローチの純粋さと教育研究分野での普及により，プログラミングというものに対する理解が高まると思われる．次世代の優秀な関数型プログラマの多くは，Haskell で最初のプログラミング経験を積むことになるだろう．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

第 9 章

全体のまとめ

7 つのプログラミング言語を学習し終えた皆さん，おめでとう．皆さんは，おそらく，「この章で言語の勝者と敗者を振り分けるのだな」と思っているかもしれないが，本書はそもそもそんなことを決めるために書いたのではない．新しいアイデアを発見してもらうために書いたのだ．筆者がまだあまりキャリアを積んでいない頃そうであったように，皆さんのなかには，大規模なチームの商用プロジェクトにどっぷりとつかって，創造力を発揮することなどまったくなく，ソフトウェア工場の一員として働いてきた方もいるだろう．そのような世界では，新しいプログラミング言語に触れる機会など著しく限られている．喩えてみれば，小さな街の映画館で，ひたすら大ヒットした映画だけを見ている 1970 年代の映画好きのようなものだ．

自分でプログラムを書くようになって以来，筆者はいつもインディーズフィルムの世界を見つけたような気持ちでいる．筆者は Ruby のコードを書いて収入を得ているが，Ruby にすべての答えがあると思うほど世間知らずではない．ちょうどインディーズフィルムによって映画製作の最先端技術が進歩するように，新しいプログラミング言語はプログラムの編成や構造に関する考え方に変化をもたらす．では，本書で学んできたことを復習してみることにしよう．

9.1 プログラミングモデル

プログラミングモデルは非常にゆっくりと変化する．歴史を振り返ると，大体 20 年くらいのサイクルで新しいプログラミングモデルが登場している．筆者が最初に覚えたのは Basic や Fortran など，いくつかの手続き型言語だった．大学では Pascal で，より構造的なアプローチを学んだ．IBM では C と C++ で商用のコードを書き始めた．Java に最初に触れたのはこの頃だ．オブジェクト指向のコードも書き始めた．筆者のプログラミング経験は 30 年以上に及ぶが，その間に出会ったプログラミングパラダイムは 2 つだけだ．読者のなかには，筆者はどうして（手続き型やオブジェクト指向以外の）新しいプログラミングパラダイムの紹介に躍起になっているのだろう，と思う人もいるかもしれない．当然の疑問だ．

プログラミングモデルは，ゆっくりとではあるが確実に変化する．そして変化が起こると，さながら竜巻が通り過ぎた後のように，プログラマはそれまで積み上げてきたキャリアを壊され，投資を惜しんできた会社はひどい打撃を受ける．プログラミングパラダイムと戦っている自分に気づいたら，注意したほうがよい．今，並行性と信頼性を求める声が高まっており，高レベルプログラミング言語への流れが起こっている．少なくとも，個々の問題を解決するためにより専門的な言語が登場するのではないかと筆者は見ている．以下，これまでに登場したプログラミングモデルを概説する．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

オブジェクト指向（Ruby，Scala）

現在の主流は，オブジェクト指向だ．その典型が Java だ．オブジェクト指向プログラミングパラダイムの主要概念は，カプセル化，継承，ポリモーフィズムの 3 つだ．Ruby では，動的なダックタイピングを経験した．Ruby では，クラスやオブジェクトの定義に基づいて契約を実施する代わりに，オブジェクトが応答できるメソッドに基づいて型付けを行う．また，コードブロックによって関数型プログラミングの概念の一部をサポートしていることも学んだ．

Scala でもオブジェクト指向プログラミングが可能だ．Scala は静的型付けをサポートしているものの，Java に比べればうるさくなく，型推論などの機能によってシンタックスの簡素化を図っている．この機能により，シンタックスと使い方をヒントにして変数の型を自動的に推論する．また Scala は，関数型プログラミングの概念の導入という点では Ruby よりも先行している．

Ruby，Scala ともに，今日，広い範囲で本稼働アプリケーションに使われており，どちらも Java などの主流言語に比べて言語設計という点で大きな進歩を遂げている．オブジェクト指向言語にはさまざまなバリエーションがある．次に挙げるプロトタイプ言語もそのひとつだ．

プロトタイプ・プログラミング（Io）

プロトタイプ言語はオブジェクト指向言語のサブセットとも言えるが，実際にはかなり異なる部分があるため，本書では異なるプログラミングモデルとして紹介した．Io ではクラスを使わずに，プロトタイプはすべてオブジェクトのインスタンスとして表現する．一部の特別に指定されたインスタンスは，他のオブジェクト・インスタンスのプロトタイプとして機能する．プロトタイプ言語ファミリーに属する言語として，JavaScript と Io がある．簡潔で表現力のあるプロトタイプ言語は通常，動的型付けを採用しており，スクリプティングやアプリケーション開発に（とりわけユーザインタフェースの開発に）適している．

Io での学習を進めていくと，シンプルなプログラミングモデルとコンパクトで一貫性のあるシンタックスの組み合わせは強力であると分かってくる．Io は，複数の連結した並行プログラムのスクリプト記述から独自の DSL のコーディングまで，広範なコンテキストで使うことができる．ただし，プロトタイプ・プログラミングよりも，さらに特殊なパラダイムが存在していた．それが，本書でも取り上げた Prolog に代表される制約論理プログラミングだ．

制約論理プログラミング（Prolog）

Prolog は，制約論理プログラミングのために構築されたプログラミング言語ファミリーに属する．本書で紹介した Prolog アプリケーションによって解いた問題の範囲はかなり狭いが，結果は目を見張るほどだった．実世界で我々が知っている論理制約を定義し，Prolog に答えを見つけさせるという形でいくつか問題を解いた．

プログラミングモデルがこのパラダイムに適合した場合は，他の言語の数分の 1 の行

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

数のコードで答えを得ることができた．このパラダイムに属する言語ファミリーは，航空管制や土木工学など，最も条件の厳しいアプリケーションで広く使われている．Prologほど洗練されてはいないが，C や Java などの言語でも論理ルールエンジンの実装例がある．Prolog は，Erlang にもインスピレーションを与えた言語である（Erlang は別の言語ファミリー，すなわち次に説明する関数型言語に属する）．

関数型プログラミング（Scala，Erlang，Clojure，Haskell）

本書で紹介するプログラミングパラダイムのなかで，最も広く期待されているのはおそらく，関数型プログラミングだ．関数型の純粋さの程度は言語によって異なるが，概念は一貫して変わらない．関数型プログラミングは，数学の関数で構成されている．同じ関数を何度呼び出しても毎回同じ結果が得られる．副作用は推奨されないか，禁止されている．こうした関数はさまざまな方法で合成できる．

各章で見たとおり，関数型プログラミング言語は通常，オブジェクト指向言語よりも表現力が豊かである．本書に掲載したサンプルプログラムは，同じ内容をオブジェクト指向言語で書いた場合よりも短くシンプルになっていることが多い．これは，オブジェクト指向パラダイムよりもプログラムを組み合わせるための広範なツールが用意されているためだ．オブジェクト指向言語では必ずしも見られない 2 つの例として，高階関数，およびやはり複雑な概念であるカリー化を紹介した．Haskell の章では，関数型としての純粋さの程度が異なると，それに応じて言語の利点と欠点も異なることを学んだ．関数型プログラミング言語の一番の利点は，副作用がないことだ．これにより並行プログラミングが容易になる．可変状態が排除されると，従来の並行性に関する問題の多くが解消される．

変化するパラダイム

関数型プログラミングをもっと試してみたいと決心したなら，さまざまなアプローチ方法がある．OOP と完全に決別することもできるし，より漸進的なアプローチを選択することもできる．

7 つの言語を学習する過程で，40 年にわたって使われている言語，および少なくとも 4 つのプログラミングパラダイムが登場した．プログラミング言語が進化していることも認識してもらえたことと思う．パラダイムを進化させるための 3 つの異なるアプローチについて見てきた．Scala のアプローチは共存方式だ．Scala のプログラマは，強い関数型の性質を用いてオブジェクト指向プログラムを構築できる．Scala では，言語自体が両方のプログラミングパラダイムを高いレベルでサポートするという方針で設計されている．Clojure のアプローチは互換性重視だ．Clojure は JVM 上に構築されているため，Clojure のアプリケーションは Java オブジェクトを直接利用できるが，「OOP は基本的に一部欠陥がある」というのが Clojure の考え方だ．Scala と違って，Clojure と Java の相互運用性は，Java 仮想マシン上の既存のフレームワークを活用することを目的として提供されており，Clojure という言語自体を拡張することが目的ではない．Haskell と Erlang は基本的にスタンドアロン言語だ．どちらも，基本的に，オブジェクト指向プログラミングを一切許容しない．このように，OOP と関数型の両方のパラダイムを受け入れる，OOP と完全に決別する，オブジェクト指向ライブラリは受け入れるが，背後にある OOP

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

パラダイムとは決別する，という 3 とおりのアプローチがある．好みのアプローチを選択すればよい．

本書で紹介したいずれかの言語を選択するかどうかは別として，プログラミング言語を取り巻く現状を知っておいたほうがよいだろう．筆者は Java の開発者として，クロージャが使えるようになるまで 10 年待たされた．これだけ待たされたのは，筆者を含め不見識なユーザが多かったため，クロージャを求める声がそれほど高くならなかったためだ．その間，Spring などの主要なフレームワークでは，クロージャを徹底的に活用できたはずの問題を解決するために無名内部クラスを多用し，身動きのとれない状況に陥っている．そのため，過酷なほどのタイプ量で指は痛み，そうした複雑なコードを読む必要があるため目を酷使する羽目になった．最近の Java 開発者は以前よりずっと知識がある．Martin Odersky や Rich Hickey といった人たちが代替案を提供してくれたからだ．こうした人たちが開発した言語によって最新技術がさらに進歩を遂げているため，Java は自分自身も進歩するか，取り残されるかの選択を迫られている．

9.2　並行性

本書で繰り返し取り上げてきたテーマの 1 つに，並行性を扱うための言語構文とプログラミングモデルの向上の必要性がある．言語によって，アプローチは大きく異なっていたが，それぞれ極めて有効であった．本書で取り上げたアプローチをここでまとめておこう．

可変状態の管理

並行性の議論で最もよく取り上げられるテーマはなんと言ってもプログラミングモデルだ．オブジェクト指向プログラミングでは，副作用と可変状態を許す．両方が同時に発生すると，プログラムは非常に複雑になる．複数のスレッドとプロセスが混在すると，複雑すぎて管理できなくなる．

関数型プログラミング言語は，構造を追加する際に前提となる重要なルールがある．同じ関数を何度呼び出しても同じ結果が返される．変数に代入できるのは 1 回だけだ．副作用がなくなると，競合状態およびそれに関連するすべての複雑さも解消される．さらに本書では，基本的なプログラミングモデルを超えた具体的なテクニックについて説明した．以下，詳しく見ていこう．

Io，Erlang，Scala のアクター

オブジェクトまたはプロセスのどちらを使うにせよ，アクターによるアプローチは同じだ．このアプローチでは，構造化されていないプロセス間通信を，アクター間の構造化されたメッセージのやり取りへと変化させる．アクターはファーストクラスの要素で，それぞれがメッセージキューを備えている．Erlang と Scala では，パターンマッチングを用いて着信メッセージを照合し，条件分岐させて実行する．第 6 章「Erlang」(p. 151)では，ロシアンルーレットを例に用いて，ゲームを実際に実行してみた．弾丸は 3 番目のシリンダーに装填したのだった．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/erlang/roulette.erl

```
-module(roulette).
-export([loop/0]).

% send a number, 1-6
loop() ->
    receive
        3 -> io:format("bang.~n"), exit({roulette,die,at,erlang:time()});
        _ -> io:format("click~n"), loop()
end.
```

次にプロセスを開始し，その際 Gun にプロセス ID を代入した．プロセスを殺すには Gun ! 3. とすればよい．Erlang の仮想マシンと言語は堅牢なモニタリング機能を備えているため，異常な兆候を最初に察知した時点で，通知したりプロセスを再起動したりできる．

フューチャ

Io ではアクターモデルに対して 2 つの並行性構文を追加している．すなわちコルーチンとフューチャだ．コルーチンを使うと，2 つのオブジェクトがお互いに適切なタイミングで制御を譲渡しながら，協調してマルチタスクで動作できる．フューチャは所要時間の長い並行計算結果のプレースホルダーであることを思い出してほしい．

サンプルでは futureResult := URL with("http://google.com/") @fetch という文を実行した．実際の結果は即座には返ってこないが，プログラムの制御は即座に復帰し，このフューチャの値を参照しようとした時点で初めてブロックする．Io のフューチャは，実際の結果値が使用可能になった時点で，その結果値で置き換えられる．

トランザクションメモリ

Clojure では，並行性に対する興味深いアプローチを多数紹介した．ソフトウェアトランザクショナルメモリ（STM）は，共有リソースに対する各分散アクセスをトランザクションでくるむ方法だ．データベースオブジェクトの場合も，これと同じ方法で並行呼び出し発生時のデータベースのデータ整合性を維持している．Clojure では各アクセスを dosync 関数でくるむ．この方法により Clojure の開発者は，必要とあれば厳密な関数型設計なしに，複数のスレッドおよびプロセス間で整合性を維持することができる．

STM は比較的新しい概念で，人気のある言語でも採用され始めている．Lisp の方言である Clojure は，こうしたアプローチには理想的な言語だ．なぜなら Lisp はマルチパラダイム言語だからだ．ユーザは，高密度の並行アクセスが生じている場合でさえ，整合性とパフォーマンスを心配することなく，必要に応じて異なるプログラミングパラダイムを使用できる．

次世代のプログラマは，プログラミング言語に対してより多くの機能を要求するようになるだろう．スレッドを開始してセマフォを待つような簡単な仕組みではもはや十分ではなくなってきている．新しく登場する言語は，並行性を実現する首尾一貫した考え方と，それに対応するツールを備えていることが求められる．並行性を求める声が高まっ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

ているため，プログラミングパラダイム全体が陳腐化する可能性もある．あるいは，変更可能な変数を厳密に制御したり，アクターやフューチャなど高機能な並行性構文を使うことで，古い言語が適応していく可能性もある．

9.3　プログラムを構成する要素

本書を執筆していて最も面白かったことのひとつは，さまざまな言語の基本構築ブロックに触れることができた点だ．言語ごとに主要な新しい概念を紹介した．ここで挙げるうちのいくつかは，読者が今後出会う新しい言語で採用されているかもしれない．ここでは筆者が特に気に入った機能を取り上げる．

リスト内包表記

Erlang，Clojure，Haskell の章で見たように[*1]，リスト内包表記は複数の概念をまとめて 1 つの強力な構文を作り上げるコンパクトな構文だ．リスト内包表記で実現可能な演算としてフィルタ，マップ，直積などがある．

最初にリスト内包表記が出てきたのは Erlang の章だ．そこでは，まず商品の入ったショッピングカート `Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}]` を例として取り上げた．このリストに消費税を追加するという問題を，簡単なリスト内包表記を作成して即座に解くことができた．

```
8>   WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
8>   {Product, Quantity, Price} <- Cart].
[{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
```

何人かの言語開発者が，リスト内包表記をお気に入りの機能の 1 つとして挙げている．その気持ちには共感できる．

モナド

筆者が最も大きく知的に成長したのはおそらくモナドの領域だろう．純粋な関数型言語では，可変状態を使ってプログラムを書くことはできない．そこで，まるで可変状態が許されているかのように問題を構築できる，そんなやり方で関数を合成できるようなモナドを作成した．Haskell には，この問題を解決するため，do 表記（モナドによってサポートされている）が用意されている．

モナドによって複雑な計算を簡素化できることも分かった．各モナドは計算戦略をサポートしている．例えば Maybe モナドを用いてエラー条件（Nothing を返す可能性のある List 検索など）を処理した．また List モナドを用いて，直積を計算したり組み合わせを解いたりした．

[*1] Scala でもリスト内包表記をサポートしているが，本書では使わなかった．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:37:31 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

マッチング

よく見かけるプログラミング機能の 1 つとして，パターンマッチングについて解説した．パターンマッチングが最初に出てきたのは Prolog の章だが，Scala，Erlang，Clojure，Haskell の章でも取り上げた．これらの言語ではパターンマッチングを用いてコードの大幅な簡素化を図っている．パターンマッチングが使われる問題領域として字句解析，分散メッセージパッシング，デストラクチャリング，ユニフィケーション，XML 処理などがある．

典型的な Erlang パターンマッチングの例として，次のような翻訳サービスを例として取り上げた．

▶ code/erlang/translate_service.erl

```
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa"} ->
            From ! "house",
            loop();

        {From, "blanca"} ->
            From ! "white",
            loop();

        {From, _} ->
            From ! "I don't understand.",
            loop()

end.

translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

loop 関数は，プロセス ID（From）とその後に続く単語（casa または blanca）またはワイルドカードに対するマッチングを行う．パターンマッチングによりプログラマは，字句解析を一切行うことなく，メッセージから必要な部分を簡単に取り出せる．

ユニフィケーション

Prolog はユニフィケーションを使用する．ユニフィケーションはパターンマッチングの親戚のようなものと考えてよい．Prolog では，左辺と右辺をマッチさせるために，ルール内の変数に入るあらゆる値を候補値がなくなるまで試す．ユニフィケーションの例として concatenate という簡単なプログラムを紹介した．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

▶ code/prolog/concat.pl

```
concatenate([], List, List).
concatenate([Head|Tail1], List, [Head|Tail2]) :-
  concatenate(Tail1, List, Tail2).
```

このプログラムがこれほどの力を備えているのは，真偽の判定，左辺のマッチング，右辺のマッチングという 3 つの操作をユニフィケーションでまかなえるからだ．

9.4　自分なりの表現を見いだす

本書では，全体を通して映画とその登場人物を引き合いに出して解説してきた．映画製作の楽しいところは，自らの経験を俳優，セット，ロケーションと溶け合わせて，自分の描きたいストーリーを紡ぎ出すところにある．すべてのエネルギーは観客を喜ばせるために注ぎ込まれる．この世界への理解が深まれば深まるほど，良い映画が出来上がる．

プログラミングについても同じように考える必要がある．プログラムにも観客がいる．といっても，アプリケーションのユーザのことを言っているのではない．コードを読む人，それがプログラムの観客だ．優れたプログラマになるには，観客に向けてコードを書き，観客を喜ばせる表現を見いだす必要がある．新しい言語の機能を学べば，そうした表現を見いだし発展させていく余裕が生まれる．表現力とは，自分自身をコードで表すその人なりの方法である．表現力の源となるのは，これまでの経験の蓄積である．本書があなたなりの表現を見いだす助けになれば幸いだ．何より，皆さんに本書を楽しんでいただけたことを願っている．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

参考資料

[Arm07] Joe Armstrong. *Programming Erlang: Software for the Concurrent World*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2007.
日本語版：Joe Armstrong 著，榊原一矢訳『プログラミング Erlang』（オーム社，2008 年）
153

[Gra04] Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly & Associates, Inc, Sebastopol, CA, 2004.
日本語版：Paul Graham 著，川合史朗監訳『ハッカーと画家 コンピュータ時代の創造者たち』（オーム社，2005 年）
190, 226

[Hal09] Stuart Halloway. *Programming Clojure*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.
日本語版：Stuart Halloway 著，川合史朗訳『プログラミング Clojure』（オーム社，2010 年）
214

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Inc., Mountain View, CA, 2008.
日本語版：Martin Odersky，Lex Spoon，Bill Venners 著，長尾高宏訳，羽生田栄一監修『Scala スケーラブルプログラミング』（インプレスジャパン，2009 年）
111

[TFH08] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.
日本語版：Dave Thomas with Chad Fowler and Andy Hunt 著，まつもとゆきひろ監訳，田和勝訳『プログラミング Ruby 1.9 言語編』『プログラミング Ruby 1.9 ライブラリ編』（オーム社，2010 年）
17

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

監訳者あとがき

あくまでも主観ですが，日本は世界でも屈指の「言語好き国家」だと思います．プログラマに言語好きが多いのは世界共通ですが，その中にあっても，日本ではプログラミング言語そのものに関心を持つ人が目立つような印象があります．毎年開かれる LL（軽量言語）カンファレンスなどを見ると，いちおう「軽量」というくくりはあるものの，特定の言語にこだわらず，言語そのものをテーマにしたカンファレンスが毎年開催され，しかも，その出席者が 1,000 名に届くような国を，私は日本以外では知りません．より正確に言うと，大学で研究としてプログラミング言語を取り上げたり，プログラミング言語をそのものを実装する人の数はそれほどでもないようですが，教養として言語に関心を持つ人の層がかなり厚い国であると言えると思います．私自身は過去 20 年間，機会あるごとに「言語の魅力」を語り続けてきましたが，仮にそれが何らかの影響を与えたとすれば，大変に名誉なことだと思います．

本書はそのような「言語好き」の人たちにとって最適の一冊であり，このような書籍が日本語で出版されるお手伝いができたことをうれしく思います．本書でも既に言い尽くされているとおり，新しい言語を学ぶことは新しい世界を広げることでもあります．本書で取り上げられた 7 つの言語，Ruby，Io，Prolog，Scala，Erlang，Clojure，Haskell はどれも特徴のある優れた言語です．それぞれの言語にはそれぞれの思想・哲学・背景・文化があります．これらの言語を学ぶことは，その言語を実際に日常的に利用するかどうかにかかわらず，プログラマとしての見識を広めてくれるに違いありません．

しかし，新しい言語の概要を理解するだけでも大変なことです．かつて，「毎年新しい言語を学ぶ」というプログラマへのチャレンジを提唱した Dave Thomas は，そのチャレンジの最初の言語として Ruby を選択し，そのまま Ruby の専門家になってしまったわけですが，数年前，彼に直接会ったときに，Ruby の次の『今年の言語』には何を選ぶつもりか尋ねたところ，「忙しくて実践できていない．が，日本語を考えている」ということでした．まあ，確かに日本語も言語ですが．

Dave Thomas でさえ実践が難しい「毎年 1 言語」ですが，本書の著者 Bruce Tate は，それよりもさらに困難な「毎週 1 言語」を実現するために，言語のエッセンスを抽出してくれました．1 週間で言語を完全にマスターすることは不可能でも，Bruce が噛み砕いてくれた概要であれば，入門としては十分以上でしょう．あくまでも概要ですが，この後，より詳細を学ぶことになっても迷子になることはないでしょう．なかなか時間がとれない「言語好き」には，またとない書籍になっていると思います．

本文中にも言及されていますが，Bruce Tate は私のホームタウン，松江に来たことがあります．2009 年に開催された第 1 回 RubyWorld Conference にスピーカーとして招待し，彼の前の著書『Java から Ruby へ』について講演していただいたのですが，そのときの短い交流からでも，彼の誠実でユーモアのあふれる温かい人柄を感じることができました．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

まさか，その後，私のような「言語好き」にぴったりの本書を執筆するとは想像もしませんでしたが．本書は彼の人柄も反映して，温かみのある文章になっています．ただし，それを日本語に反映することは大変困難でした．実際の翻訳作業を担ってくださった田和さんに心から感謝します．また，技術的な間違いの指摘，および訳文の品質向上に貢献してくださったレビューアのみなさんに感謝します．榊原一矢さん，西尾泰和さん，林拓人さん，井上真さん，和田卓人さん，関将俊さん，三村益隆さん，舞波さん，ありがとうございます．みなさんの力がなければ本書の品質はここまで到達しなかったでしょう．

本書が読者のみなさんの世界を広げる一助になりますように．また，言語好きが世に栄えますように．

期待をこめて．

2011 年 7 月

まつもと ゆきひろ

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

索引

A

Armstrong, Joe 151, 153–154

C

Clojure 3, 189–229
- Java との統合 190, 213, 224, 227
- leiningen 191
- Lisp と── 190, 226, 227, 229
- repl 191
- アトム 221–222
- インストール 191
- エージェント 222–224
- 型付けモデル 193
- 関数 200–204
- コンソール 191
- 再帰 207–208
- 作者 204–206
- 算術演算 192–193
- 参照 219–220
- シーケンス 208–213
- 式 194–196
- スレッドの状態 225
- セット 197–198
- 前置記法 228
- 遅延評価 211–213, 228
- 強み 227–228
- デストラクチャリング 201
- トランザクションメモリ 219–220, 275
- バインディング 201–202
- パターンマッチング 277
- ブール値 194–196
- フォーム 193
- フューチャ 224, 275
- プログラミングモデル 273
- プロトコル 213–216
- 並行性 219–224, 225, 227
- ベクタ 197
- マクロ 216–218
- マップ 198–199
- マルチメソッド 225
- 無限数列 211–213
- 無名関数 202–204
- メタデータ 224
- 文字列 193–194
- 読みやすさ 229
- 弱み 228–229
- リスト 196, 276

clone メッセージ（Io における） 43
Colmerauer, Alain 74

D

Dekorte, Steve 41, 52–53

E

Erlang 3, 151–187
- アクター 152, 274–275
- アトム 157–158
- 型付けモデル 160
- 関数 160–163, 166–169
- 繰り返し 167
- 軽量プロセス 152, 185
- 作者 151, 153–154
- 式 156–157
- 信頼性 152, 153, 179–183, 185
- 制御構文 164–166
- タプル 157–158
- 短所 186
- 強み 185–186
- パターンマッチング 158–160, 277
- プログラミングモデル 155, 273
- 並行性 152–153, 174–179
- 変数 156–157
- 無名関数 166
- メッセージ 174–179
- ライブラリ 185
- リスト 157–158, 166–173, 276
- リンクされたプロセス 179–183

F

foldLeft メソッド（Scala における） 136
forward メッセージ（Io における） 65–66

H

Haskell 4, 231–270
- 型 232–235, 254–259
- 型付けモデル 231, 268
- カリー化 248–249
- 関数 235–236, 256–257
- 関数合成 239–240
- 起動 232
- クラス 259
- 高階関数 246–248
- 再帰 236–237, 241–242
- 再帰型 257–259
- 作者 244–245, 251–253
- 式 232–235
- 畳み込み 247–248
- タプル 237–240
- 遅延評価 249–251
- 強み 268–269
- パターンマッチング 277
- 範囲 242–243
- フィルタリング 247–248
- 部分適用関数 247–249
- プログラミングモデル 231, 268, 269, 273
- ポリモーフィズム 256–257
- マッピング 246–247
- 無名関数 246
- モナド 260–266, 276
- リスト 240–243, 276
- リスト内包表記 243

Hickey, Rich 204–206

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

I

Io 3, 41–72
- clone メッセージ 43
- forward メッセージ 65–66
- アクター 68, 274–275
- インストール 42
- インタープリタ型モデル 42
- 演算子 56–57
- オブジェクト 42–46, 48
- オブジェクトのスロット 43, 48
- 型付けモデル 45
- 繰り返し 54–56
- 継承 44–46
- 作者 41, 52–53
- 条件 50–52
- 条件文 54–56
- 代入 57
- 強み 70–71
- ドメイン固有言語（Io による） 62–64
- パフォーマンス 72
- フューチャ 69, 275
- プログラミングモデル 272
- プロトタイプ 42, 48
- 並行性 66–69, 71
- マップ 49
- メソッド 46–48
- メッセージ 42, 48, 58–60
- 弱み 71–72
- リスト 48–50
- リフレクション 60
- ループ 55–56

J

Java
- Clojure と── 190, 213, 224, 227
- Scala と── 110, 147

L

Lisp（Clojure の祖先としての） . 190, 226, 227, 229

M

method_missing の振る舞い（Ruby における） 32–33

mixin（Ruby における） 26–29

O

Odersky, Martin 111

P

Peyton-Jones, Simon 251–253

Prolog 3, 73–108
- 8 クイーンの例 100–104
- append ルール 91–94
- 大文字（Prolog における） 75
- 再帰 85–87, 90–91
- 作者 74
- 質問 77–78
- 推論 76–77
- 数値計算 90–91
- 数独の例 96–100
- スケジュール作成の例 82–84
- タプル 87–90
- 知識ベース 74, 75
- 地図の塗り分けの例 78–80
- 強み 106–107
- パターンマッチング 277
- プログラミングモデル 272
- ユニフィケーション 80–82, 87–90, 277
- 弱み 107
- リスト 87–91

R

Roussel, Phillipe 74

Ruby 2, 9–40
- method_missing の振る舞い 32–33
- mixin 26–29
- Web 開発 38
- インストール 11
- インタープリタモデル 12
- 宇宙船演算子 28
- オープンクラス 31–32
- 型付けモデル 16–17
- 型の安全性 40
- 関数 18
- クラス 24–26
- コードブロック 21–23
- 作者 9–10
- 市場までのリードタイム 39
- スクリプティング 38
- 対話型コンソール 12
- 強み 38–39
- 配列 18–20
- ハッシュ 20–21
- パフォーマンス 39
- 判定構文 13–15
- ファイルからの実行 23
- プログラミングモデル 12, 272
- 並行性 40
- メタプログラミング 30–37
- モジュール 26–29, 33–37
- 弱み 39–40
- 歴史 9
- 文字列の処理 12

S

Scala 3, 109–149
- Any クラス 131
- foldLeft メソッド 136
- Java と── 110, 147
- Nothing 型 131
- XML と── 138–139, 140, 148
- アクター 141–145, 274–275
- 型 113–114
- 型付けモデル 115, 148
- カリー化 136
- 関数 131–137
- クラス 120–125
- 繰り返し 116–118, 132–134
- 継承 123
- 高階関数 131
- コレクション 127–137
- 再帰 134
- 作者 111

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

式 114–116
条件 114–116
セット 129–130
タプル 119–120
強み 146–148
ドメイン固有言語（Scala による） 147
トレイト 124
パターンマッチング 139–141, 277
範囲 118–120
プログラミングモデル 272, 273
並行性 112, 127, 141–145, 147
変更不能な変数 127
補助コンストラクタ 122
マップ 130–131
弱み 148–149
リスト 128–129, 132–137

T
Tarbox, Brian 84

W
Wadler, Philip 244–245
Web 開発（Ruby による） 38

X
XML（Scala での処理） 138–139
XML（Scala におけるサポート） 148
XML（Scala によるパターンマッチング） 140

ア
アクター 274–275
  Erlang 152
  Io 68
  Scala 141–145
アトム（Clojure における） 221–222
インタープリタ型言語
  Ruby 12
  ――としての Io 42
宇宙船演算子（Ruby における） 28
エージェント（Clojure における） 222–224
オープンクラス（Ruby における） 31–32
オブジェクト
  Io 48
オブジェクト指向言語 272
  Io（オブジェクト指向言語としての） 42
  Ruby 12
  Ruby（オブジェクト指向言語としての） .. 272
  Scala（オブジェクト指向言語としての） .. 272
オブジェクトのスロット（Io における） 43, 48

カ
型付けモデル 1
  Clojure 193
  Erlang 160
  Haskell 268
  Io 45
  Ruby 16–17
  Scala 113–114
型の安全性（Ruby における） 40
可変状態（の管理） 274
カリー化
  Haskell 248–249
  Scala 136
関数
  Clojure 200–204
  Erlang 160–163, 166–169
  Haskell 235–236, 256–257
  Ruby 18
  Scala 131–137
関数型プログラミング言語 273–274
  Clojure（関数型プログラミング言語としての） 206
  Erlang 155
  Haskell 231, 268, 269
  Scala 126–127
  並行性と―― 112
クラス
  Haskell 259
  Ruby 24–26
  Scala 120–125
繰り返し
  Erlang 167
  Io 54–56
  Scala 116–118, 132–134
継承
  Io 44–46
  Scala 123
軽量プロセス（Erlang における） 185
ゲーム（Prolog による） 106
高階関数
  Haskell 246–248
  Scala 131
コードブロック（Ruby における） 21–23
コンパイル型言語
  Scala 114
  ――としての Prolog 74
コンパイル言語
  Erlang 156

サ
再帰
  Clojure 207–208
  Haskell 236–237, 241–242
  Prolog 85–87, 90–91
  Scala 134
再帰型（Haskell における） 257–259
参照（Clojure における） 219–220
シーケンス（Clojure における） 208–213
式（Haskell における） 232–235
自然言語処理（Prolog による） 106
質問（Prolog における） 77–78
条件
  Io 50–52
  Scala 114–116
シンタックスシュガー 9
推論（Prolog における） 76–77
スクリプティング（Ruby における） 38
スレッドの状態（Clojure における） 225
静的型付け
  Haskell 231
  Scala 115, 148
セット
  Clojure 197–198

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

Scala 129–130
セマンティック Web（Prolog による） 106
宣言型言語（としての Prolog） 73

タ

第一級 13
対話型モデル 2
畳み込み（Haskell における） 247–248
ダックタイピング（Ruby における） 16–17
タプル
Erlang 157–158
Haskell 237–240
Prolog 87–90
遅延評価
Clojure 228
Haskell 249–251
知識ベース（Prolog における） 74, 75
強く型付けされた言語
Clojure 193
Erlang 163
Haskell 231, 268
Ruby 16
Scala 114
データ構造体 2
動的型付け
Clojure 193
Erlang 160, 163
Ruby 16
ドメイン固有言語
Io による 62–64
Scala による 147
トランザクションメモリ（Clojure における） 219–220, 275
トレイト（Scala における） 124

ハ

ハイブリッド言語（としての Scala） 109–110
配列（Ruby における） 18–20
パターンマッチング 277
Erlang 158–160
Scala 139–141
ハッシュ（Ruby における） 20–21
パフォーマンス
Io 72
Ruby 39
範囲（Haskell における） 242–243
ファーストクラス 13
フィルタリング（Haskell における） 247–248
フォーム（Clojure における） 193
部分適用関数（Haskell における） 247–249
フューチャ 275
Clojure 224
Io 69
プログラミング言語
インストール 6
学ぶ 1–2, 4–5
プログラミングモデル 1, 271–274
プロトコル（Clojure における） 213–216
プロトタイプ（Io における） 48
プロトタイプ・プログラミング言語（としての Io） 272
プロトタイプベースの言語（としての Io） 42
分岐構文 2
並行性 274–276
Clojure 219–224, 225, 227
Erlang 152–153, 174–179
Io 66–69, 71
Ruby 40
Scala 112, 127, 141–145, 147
ベクタ（Clojure における） 197
ポリモーフィズム（Haskell における） ... 256–257

マ

マクロ（Clojure における） 216–218
マップ
Clojure 198–199
Haskell 246–247
Io 49
Scala 130–131
まつもとゆきひろ（Matz） 9–10
マルチメソッド（Clojure における） 225
無名関数
Clojure 202–204
Erlang 166
Haskell 246
命令型言語 73
メソッド（Io における） 46–48
メタデータ（Clojure における） 224
メタプログラミング（Ruby における） 30–37
メッセージ（Io における） 42, 48, 58–60
モジュール（Ruby における） 26–29, 33–37
文字列
Clojure 193–194
Ruby での処理 12
モナド（Haskell における） 260–266, 276

ヤ

ユニフィケーション（Prolog における） 80–82, 87–90, 277

ラ

リスト 276
Clojure 196
Erlang 157–158, 166–173
Haskell 240–243
Io 48–50
Prolog 87–91
Scala 128–129, 132–137
リフレクション（Io における） 60
リンクされたプロセス（Erlang における） 179–183
論理プログラミング言語（としての Prolog） 73, 272

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:43:45 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

著者

Bruce A. Tate（ブルース・テイト）

テキサス州オースティンに本拠地を置く RapidRed を経営する．RapidRed は Ruby でライトウェイトな開発を行うコンサルティングファーム．過去に IBM でデータベースシステムプログラマや Java プログラマなどとして勤務．その後，ベンチャー企業数社でクライアントソリューション担当部長や CTO などを勤める．米国外での講演も多数．著書は 10 冊を超え，*From Java to Ruby*，*Deploying Rails Applications* やベストセラーの “Bitter” シリーズ，*Beyond Java*，ジョルト賞を獲得した *Better, Faster, Lighter Java* などがある．

監訳者

まつもとゆきひろ

株式会社ネットワーク応用通信研究所フェロー．その他，肩書多数．Ruby の生みの親．三女一男の父であり，良き夫でもある．性格はふまじめだが，敬虔（？）なキリスト教徒である．温泉好き．鳥取県出身，島根県在住．牡牛座．O 型．

訳者

田和勝（たわまさる）

フリーランス翻訳者．IT 関連書籍．ドキュメントの翻訳を専門に行っている．主な訳書として，『Perl クックブック』『詳説 正規表現 第 2 版』（いずれもオライリージャパン刊），『プログラミング Ruby 1.9 言語編』『同ライブラリ編』（オーム社刊）などがある．兵庫県神戸市出身．

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from “Maruzen eBook Library” on 12/05/2020 11:50:28 under the license of University of Tokyo
7つの言語7つの世界 ―Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell― [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:50:28 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社

May not be copied, displayed, distributed, modified, published, reproduced, stored, transmitted all or any part of the content
from this site in any medium to anyone except for personal and non-commercial use permitted under the copyright law of Japan.

- 本書の内容に関する質問は，オーム社開発部「7 つの言語 7 つの世界」」係宛，E-mail（kaihatu@ohmsha.co.jp）または書状，FAX（03-3293-2825）にてお願いします．お受けできる質問は本書で紹介した内容に限らせていただきます．なお，電話での質問にはお答えできませんので，あらかじめご了承ください．
- 万一，落丁・乱丁の場合は，送料当社負担でお取替えいたします．当社販売管理課宛お送りください．

eBook Library

7 つの言語 7 つの世界

2011 年 7 月 25 日　第 1 版 第 1 刷 発行
2012 年 2 月 6 日　第 1 版 第 3 刷 発行（20120206, 959）

著　者　Bruce A. Tate
監訳者　まつもとゆきひろ
訳　者　田和勝
企画編集　オーム社 開発局
発行人　竹生修己
発行所　株式会社オーム社
郵便番号 101-8460
東京都千代田区神田錦町 3-1
電話 03-3233-0641（代表）
URL http://www.ohmsha.co.jp/

© オーム社 2011

装丁デザイン：トップスタジオデザイン室（轟木亜紀子）
ISBN978-4-274-06857-7

複製禁止／Downloaded from "Maruzen eBook Library" on 12/05/2020 11:50:28 under the license of University of Tokyo
7つの言語7つの世界 —Ruby，Io，Prolog，Scala，Erlang, Clojure and Haskell— [2011] オーム社